Sixth Framework Programme
Information Society Technology

RE-TRUST

Remote EnTrusting by RUtime Software authentication

Project Number: FP6 - 021186

Deliverable: D3.4

HW assisted SW protection for entrusting
# Summary

<table>
<thead>
<tr>
<th>Summary</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Number:</td>
<td>FP6-021186</td>
</tr>
<tr>
<td>Project Title:</td>
<td>RE-TRUST: Remote EnTrusting by RUN-time Software authentica</td>
</tr>
<tr>
<td>Deliverable Type:</td>
<td>RP</td>
</tr>
<tr>
<td>Deliverable Number:</td>
<td>D3.4</td>
</tr>
<tr>
<td>Contractual Date of Delivery:</td>
<td>September 2009</td>
</tr>
<tr>
<td>Actual Date of Delivery:</td>
<td>September 2009</td>
</tr>
<tr>
<td>Title of Deliverable:</td>
<td>Hardware Assisted Software Protection for Entrusting</td>
</tr>
<tr>
<td>Workpackage Contributing to the Deliverable:</td>
<td>WP3</td>
</tr>
<tr>
<td>Nature of the Deliverable:</td>
<td>Report</td>
</tr>
<tr>
<td>Author(s):</td>
<td>Brecht Wyseur (K.U.Leuven)</td>
</tr>
<tr>
<td>Reviewer(s):</td>
<td></td>
</tr>
</tbody>
</table>

**Abstract:** This document presents a final overview of the techniques that have been developed in WP3: software protection techniques that utilize light-weight hardware.

**Keywords:** Software protection, remote entrusting, remote attestation, Smart Dongle, Trusted Platform Module, FPGA, Barrier Slicing.

**Classification:**

<table>
<thead>
<tr>
<th>Name of Client:</th>
<th>European Commission</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distribution List:</td>
<td>European Commission Project Partners</td>
</tr>
<tr>
<td>Authorised by:</td>
<td></td>
</tr>
<tr>
<td>Issue:</td>
<td>1.0</td>
</tr>
<tr>
<td>Reference:</td>
<td>RE-TRUST Deliverable</td>
</tr>
<tr>
<td>Total Number of Pages:</td>
<td>--</td>
</tr>
</tbody>
</table>
| Contact Details: | Prof. Yoram OFEK  
University of Trento, Dept. of Information and Communication Technology (DIT), Trento, Italy  
Phone: +39 0461 88 3954 (office)  
Fax +39 0461 88 2093 (DIT)  
E-mail: ofek -at- dit.unitn.it |
# Table of contents

1. Publications covered by this document ......................................................... 4
2. Introduction ................................................................................................ 5
3. Techniques .................................................................................................. 7
   3.1 TPM-based remote software attestation on legacy operating systems .......... 7
   3.2 FPGA-based approach ......................................................................... 8
   3.3 Barrier slicing with Smart Card ............................................................ 8
   3.4 Smart Dongle ..................................................................................... 9
4. References .................................................................................................. 10
1. Publications covered by this document

The purpose of this deliverable is to cover the techniques that have been developed in WP3 of the RE-TRUST project. Since most of these techniques have been published and presented at conference, the purpose of this report is mainly to assemble the following publications.


2. Introduction

Task 3.2 of the RE-TRUST investigates how a small amount of trusted hardware can be used to address the ‘remote entrusting’ problem. In general, we distinguish two approaches here:

1. how can hardware be used to strengthen the software-only protection techniques that were developed in WP2, and
2. the development of new solutions that rely on the presence of trusted hardware (and for which no pure-software counterpart has been developed).

With respect to the first approach, we note that during this project, we have identified security issues in software-only solutions. For example, during our discussions at project meetings, it became clear that remote software attestation techniques such as the Pioneer system ([SeL+05], and discussed intensively in [A1,A2]) suffer from network delay and hence become infeasible to deploy on generic architectures such as the RE-TRUST project envisions. Hence, we have developed a strengthened solution that uses a functionality of the Trusted Platform Module (TPM) in [A1,A2].

On the other hand, we have also developed completely new solutions for which no pure software counterpart has been developed before, such as the FPGA approach which verifies the integrity of run-time execution by means of direct access to the memory pages of the untrusted host.

The hardware that was envisioned in this task is lightweight hardware that is widely available. The purpose of this is to enable large-scale deployment. In this project, we aimed to develop solutions that use the following three types of lightweight hardware:

A Smart Card – An integrated chip that typically contains volatile memory and a microprocessor. Such chips are typically present in credit cards or cell phone SIM cards. In this project, the chip is assumed to be tamper resistant and hence can be used as a place for secure storage (of keys in the volatile memory) and secure computations.

A TPM – A Trusted Platform Module (TPM) is a secure cryptographic co-processor that can store cryptographic data (keys) and is fixed to a motherboard. Many computing devices that are sold these days are equipped with a TPM, and there are a number of Trusted Computing (TC) initiatives (such as in the OpenTC project) to develop secure computing systems based on such TPMs. However, these initiatives suffer from the need for very complex systems (secure operating systems, virtualization, and so forth) and as a result or not yet widely adopted. Moreover, in the case that these systems are deployed, in the general case they are only able to guarantee load-time verification of applications. The RE-TRUST project can be seen as a counterpart for these initiatives in the sense that we aim to develop run-time verification techniques without the need for the complex infrastructures that TC-enabled systems require.

An FPGA – A Field Programmable Gate Array is an integrated circuit that consists of a large number of gates that can be re-configured. They can be used to implement logical functions that an ASIC would otherwise perform. The main advantage is the ability to modify their functionality, which is what we intend to exploit in our solutions.

The expected outcome of this task is the development of a number of hardware assisted software protection mechanisms. In Deliverable 3.2 of this project, we have presented an intermediate report on the progress of the development of the solutions. Since this intermediate report, we had a substantial progress in this task. We developed new solutions, performed an increased security analysis of the preliminary techniques we had developed and improved them, and published our new results at international conferences and in
international journals. In this deliverable, we show that we succeeded in the aim of this task by presenting four solutions in this deliverable.

The organization of this deliverable is to present an executive summary of each of these solutions. Since all these solutions have been published and presented at international conferences, we remain with this executive summary, and have included the respective publications as an appendix to this deliverable.
3. Techniques

We present an overview of the hardware-assisted software protection techniques that have been developed.

3.1 TPM-based remote software attestation on legacy operating systems

In [A1, A2], Schellekens et al. presented a TPM-based remote software attestation technique that was developed within the context of the RE-TRUST project. Both publications are attached to this deliverable. Publication [A1] is presented at an international conference, after which the work was further improved and a journal version was accepted [A2].

Summary. In the past, several solutions towards remote code integrity verification have been developed. This problem is the act of delivering attestations to a verification entity that guarantee code executes untampered on a remote untrusted computing platform. This 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. So far, establishing a trusted execution environment on an untrusted platform has been an open research problem.

Seshadri et al. [SeL+05, SeL+06] proposed the Pioneer system, which establishes whether software on an untrusted host is untampered by calculating a checksum over its memory pages. 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 and legacy Personal Computer systems, but it relies on some strong assumptions. To determine the expected execution time of the checksum computation, detailed knowledge about the processor of the untrusted platform is needed. The expected execution time can also be unreliable because the verifier has to make a worst case assumption on the network latency, which can be rather unpredictable on the Internet.

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 that gets loaded during startup of the program, and can later report its configuration to a remote entity with an attestation protocol [SaSt04]. Unfortunately, this technology requires a secure operating system. If legacy operating systems are used, the chain of trust can easily be subverted.

In this work, we have proposed a mixed solution that addresses most of the issues raised in the pure software solution and the trusted computing approach. We rely on the TPM for trusted time measurements of the execution of the function that calculates a checksum over the memory pages, while we do not require the deployment of a secure operating system. Our solution is further extended with a trusted bootloader to be able to perform a benchmark time measurement, and to report on the system hardware of the untrusted computing platform.


3.2 FPGA-based approach

The solution developed by Basile et al. [A3] exploits the use of reconfigurable computing coupled with code mobility to build a consistent design methodology for remote code integrity verification. The term reconfigurable computing refers to systems incorporating hardware programmability enabling a distinction between physical and logical hardware, as well as run-time hardware reconfigurability (e.g., Field Programmable Gate Arrays – FPGA). This solution tries to join the benefits in terms of security of hardware based verification, together with the flexibility of mobile software techniques.

The trusted server performs its verification by observing a set of attestation received from the program that is running on the untrusted host. These attestations are generated in a challenge-response protocol between the server and the FPGA. The FPGA is in charge of computing the attestations by means of direct access to the host main memory, thus bypassing any need of interaction with the original program or with the operating system that might be under control of the adversary. Note that the monitoring facility in the FPGA is not a static component, but sent by the server in the form of a mobile hardware agent. This allows for periodically replacement of this component, thus reducing the ability of the adversary to mount attacks on a large scale.


3.3 Barrier slicing with Smart Card

This technique is an extension of the software-only technique [CeP+07] that was developed in WP2. The idea is based on a technique introduced by Zhang and Gupta [ZhGu03] to move tamper-sensitive parts of client computations to a server. Copy-sensitive parts of the program that is intended to run on the untrusted platform are sliced and moved to the server to make copying ineffective. The barrier-slicing solution in [CeP+07] focuses on invalid state and moves the invalid sensitive part of the client state to the server. The disadvantage of these solutions is that the trusted server, which is the only reliable source of trust in the software-only architecture, is overloaded with computations that cannot be performed safely on the untrusted platform, which limits its practical large-scale usage.

In this result, the use of hardware allows to improve the trade-off between security and efficiency. A tamper-resistant smart card is used to offload critical code onto. As a result, the core of trust can be split between the trusted server, and the local lightweight hardware.

In their work, Ceccato et al. [A4] present a comparative analysis on the performance and security between the proposed distributed system to the original centralized implementation on a case study. They show that this improved technique dramatically lowers the network traffic, and requires smaller memory and computational requirements to the trusted server. Certainly in large-scale applications with a high ratio of clients per trusted server, such as for example in grid computing or online-gaming scenarios, this approach proves to be valuable.

3.4 Smart Dongle

The smart dongle case is somehow different from the other techniques in the sense that the hardware that is used is not intended to leverage the techniques that were developed in WP2. Rather, a Smart Card has the role of the trusted server, and provides the service to the application. In the application that Aussel et al. [A5] present, the application is a Voice-over-IP (VoIP) client, where the smart card provides the service to connect to the network of a telephone operator. In this example, it is necessary that the client software that is executed on an untrusted platform is executing correctly to prevent malware to perform VoIP calls or digitally sign documents without the user consent and knowledge.

The smart card is embedded into a USB dongle (coined Smart Dongle), together with flash memory and a controller that provides the interface to the untrusted host. The original application (the VoIP client), resides on a CDROM partition on the Smart Dongle. Since all applications on the CDROM partition are read-only, it is not possible to tamper with these applications. However, an attack that might be possible is that the application is tampered with at load-time or at run-time when it is executing on the untrusted host. This is what we intend to prevent.

When the dongle is connecting to the untrusted platform, session keys are generated on the dongle that enable the establishment of a secure channel between the application that will run on the untrusted host, and the smart card. The keys on the smart card reside in a protected environment; the keys at the other end of the channel are embedded into the application using white-box cryptography, in order to make it hard for the adversary to reveal these keys and break into the secure channel.

A thumbprint mechanism is deployed to provide external authentication when session keys need to be renewed. This functions as a pure software verification mechanism, as has been developed in WP2 of the project, where hashes are computed of code segments, and compared to pre-computed thumbprints that are stored in a table on the smart card.

Note that in Deliverable 3.2 of this project, we have described a detailed technical overview of the Smart Dongle. We refer to this deliverable, and to the publication [A5] that is included as an appendix for further details on this protection mechanism.

4. References


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

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 cannot 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).

1 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).

1571-0661/$ – see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.entcs.2007.10.014
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 2.

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 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:\[ PCR_{\text{new}} \leftarrow \text{SHA-1}(PCR_{\text{old}}||M) \] with \( PCR_{\text{old}} \) the previous register value, \( PCR_{\text{new}} \) 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\(^4\). 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.

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.

\(^3\) 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.

\(^4\) TrustedGRUB (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\(^1\), 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: $t_1 \leftarrow t_{\text{current}}$.

(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: $c \leftarrow \text{checksum}(n,V)$.

(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 ($t_2 \leftarrow t_{\text{current}}$),
i.e., the verifier knows an upper bound on the expected execution time of the checksum calculation: \( t_2 - t_1 < \Delta t_{\text{expected}} = \Delta t_{\text{cksum}} + \Delta t_{\text{network}} + \delta t \), with \( \Delta t_{\text{cksum}} \) the expected execution time of the checksum function, \( \Delta t_{\text{network}} \) 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: \( h \leftarrow \text{hash}(n, E) \).

(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.

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 \( \text{cksum}() \) 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 \texttt{ TPM_TickStampBlob} command to create a time stamp on a blob: \( TS \leftarrow \text{Sign}_{SK}(blob|t||TSN) \) 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.

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 \( \mu \)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. Subsequent invocations of the \texttt{ 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\(^5\). 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

\(^5\) 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](image)

Other commands between two \texttt{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


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

A R T I C L E   I N F O

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

A B S T R A C T

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 cannot provide strong security assurance. We present a mixed solution by using the trusted computing hardware, namely the timestamping 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.

© 2008 Elsevier B.V. All rights reserved.

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...
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 cannot 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 cannot 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**\(^2\): \[ PCR_{\text{new}} \leftarrow \text{Hash}(PCR_{\text{old}} || M) \] with \(PCR_{\text{old}}\) the previous register value, \(PCR_{\text{new}}\) 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)\(^4\) 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](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: $t_1 \leftarrow t_{\text{current}}$. 
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: \( c \leftarrow \text{cksum}(n, V) \).

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 \( (t_2 \leftarrow t_{\text{current}}) \), i.e., the verifier knows an upper bound on the expected execution time of the checksum calculation:
\[
    t_2 - t_1 < \Delta t_{\text{expected}} = \Delta t_{\text{cksum}} + \Delta t_{\text{network}} + \delta t
\]
with \( \Delta t_{\text{cksum}} \) the expected execution time of the checksum function, \( \Delta t_{\text{network}} \) 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.

The verification agent computes a cryptographic hash of the executable \( E \) as a function of the original nonce: \( h \leftarrow \text{Hash}(n, E) \).

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.

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 \( \text{cksum}() \) 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 timestamping

Timestamping is one of the new features in version 1.2 of the TPM specification. The TPM can create a time stamp on a blob: \( TS \leftarrow \text{Sign}_{SK}(\text{blob}||t||\text{TSN}) \) with \( SK \) a signature key, \( \text{blob} \) the digest to stamp, \( t \) the current time and \( \text{TSN} \) 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 \( \text{TSN} \). 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 \( \text{TSN} \) 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 \( 1 \mu s \), 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 \text{TPM\_GetT\_c\_c\_a\_s} 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: \( TS_1 \leftarrow \text{Sign}_{SK}(n||t_1||\text{TSN}_1) \). 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: \( c \leftarrow \text{cksum}(TS_1, V) \).
4. The calculated checksum gets time stamped by the TPM as well: \( TS_2 \leftarrow \text{Sign}_{SK}(c||t_2||\text{TSN}_2) \). 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 \( \text{TSN}_1 = \text{TSN}_2 \) (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 \text{TPM\_GetT\_c\_c\_a\_s} requests, confirms that this is the tick counter increment on every command.
(c) extract $t_2 - t_1$ from the time stamps and check whether it corresponds with the expected execution time of the checksum function:

$$t_2 - t_1 < \Delta t_{\text{expected}} = \Delta t_{\text{checksum}} + \Delta t_{\text{sign}} + \delta t$$

with $\Delta t_{\text{checksum}}$ 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.

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 $\Delta t_{\text{sign}}$, 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 $\Delta t_{\text{expected}}$ 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., $\Delta t_{\text{checksum}}$). It should be noted that the length of the pseudo-random walk calculated by $\text{checksum}(\cdot)$ 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 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:

- **TickCountValue**: The number of ticks since the start of this tick session, represented with a 64 bit counter.
- **TickIncrementRate**: The rate at which the tick counter is incremented expressed in $\mu$s.
- **TickSessionNonce**: 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

/*
 * \rief 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) 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, 0x0f1};

  i = tpm_transmit(buffer, 10, 42);

  printf("TPM_GetTicks()\n");
  printf("Tag:\t%x\n", htonl(*short *(buffer+10)));
  printf("Count:\t%llx\n", *(unsigned long long *)(buffer+12));
  printf("Rate:\t%lx\n", htonl(*short *(buffer+20)));
  printf("Nonce:\t\n");
  for (i=0; i<20; i++)
    printf("%2x", buffer[22+i]);
  printf("\n");
ERROR(12, (10 != i && i != 42), "could not send to the TPM");
ERROR(13, 0xC400 == *(unsigned short *) buf, "response code mismatch");
ERROR(15, 0 == *(unsigned *) (buf+6), "error: %d", htonl(*(unsigned *) (buf+6)));
ERROR(14, 0x2a000000 == *(unsigned *) (buf+2), "size mismatch %x", *(unsigned *)(buf + 2));
return 0;

References


MobHaT: code mobility and reconfigurable computing joining forces for software protection

Cataldo Basile, Member, IEEE, Stefano Di Carlo, Member, IEEE, and Alberto Scionti, Member, IEEE,

Abstract—The explosive growth of computer networks, together with the massive use of embedded systems from low-end to high-end devices, make ubiquitous and pervasive computing a reality. However, there are still a number of new challenges to its widespread adoption that include increased need for scalability, availability, and especially security of software. Among the different challenges in software security, the problem of remote code integrity verification is still waiting for efficient solutions. This paper proposes the use of mobile hardware agents implementing monitoring techniques for securely producing and delivering attestations of code integrity of an application. It tries to join the benefits in terms of security of hardware based verification, together with the flexibility of mobile software techniques. The proposed solution perfectly targets embedded computational nodes that are nowadays commonly equipped with reconfigurable hardware components exploited for solving different problems.

Index Terms—Software protection, attestation, reconfigurable computing.

I. INTRODUCTION

The explosive growth of computer networks, together with the massive use of embedded systems from low-end to high-end devices, make ubiquitous and pervasive computing a reality. However, there are still a number of new challenges to its widespread adoption that include increased need for scalability, availability, and security of software [1]. Security issues are a serious problem and attacks against these systems are becoming more critical and sophisticated.

When considering software security, there are two key challenges that arise in a distributed computing environment. First, each computational node (e.g., PDAs, cellphones, laptops, etc.) may wish to be sure that any software deployed through the network has not been modified to perform malicious actions. Second, computational nodes that collaborate over the network, may want to be sure that only legitimate untampered applications running on remote parties have part in the collaboration. Hence, they want to verify if remote software is running untampered on a remote untrusted node.

While cryptographic signature schemes and sandboxed clients largely address the first of these concerns [2], [3], the second one still remains an open problem. A verification entity can verify the integrity of an application running on a remote untrusted host by means of attestations (proofs) proving the integrity of the executed code. Based on this concept, the problem of remote code integrity verification, representing the target of this paper, can be defined as the act of computing, and delivering such attestations to the verification entity that, by observing the received proofs, should be able to understand whether the target code is running untampered or not [4]. On the target execution platform, an adversary has administrative privileges and can tamper with all available software including the operating system.

This paper exploits the use of mobile hardware agents coupled with code mobility to build a consistent design methodology for remote code integrity verification. We consider here an extended notion of mobility taking into consideration the possibility of migrating logical hardware components in a distributed infrastructure.

Code mobility is nowadays widely used to build complex distributed systems. Systems supporting mobility are inherently dynamic and are particularly advantageous for networked infrastructures made of distributed and heterogeneous nodes [5], [6], [7]. The idea of code mobility is however mainly limited to software. This is somewhat obvious when looking at the traditional partitioning of computers into physical, static hardware and logical, mobile software. However, such partitioning is being more and more blurred by the emerging field of reconfigurable computing [8]. The term reconfigurable computing refers to systems incorporating hardware programmability enabling a distinction between physical and logical hardware, as well as run-time hardware reconfigurability (e.g., Field Programmable Gate Arrays - FPGA). Reconfigurable computing, coupled with code mobility opens up a range of new opportunities for distributed digital systems, including the possibility of achieving adaptability, functionality extension, dependability, and reliability [9], [10], [11], [12].

This paper proposes the use of mobile hardware agents implementing software monitoring techniques for securely producing and delivering attestations of the code integrity of an application. It tries to join the benefits in terms of security of hardware based verification, together with the flexibility of mobile software techniques. The proposed solution perfectly targets embedded computational nodes that are nowadays commonly equipped with reconfigurable hardware components exploited for solving different problems [8]. Finally, the proposed infrastructure has been already exploited in [11] for increasing the dependability of embedded systems. It thus provides an interesting solution to increase both security and dependability of distributed systems.

The paper is organized as follows: section II introduces the target scenario with particular attention to the attackers capabilities, while section III overviews state-of-the-art techniques for software integrity protection. The proposed
protection schema is detailed in section IV, while section V proposes an implementation of hardware monitors together with experimental data to be used to analyze feasibility and scalability. Finally, section VI proposes a security analysis trying to highlight strengths and weakness of the solution, and section VII summarizes the main contributions of the paper highlighting future activities and improvements.

II. REMOTE CODE INTEGRITY VERIFICATION PRINCIPLES

This paper considers a scenario composed of a network of computational nodes (hosts), each one able of running software applications (programs). Computational nodes comprise two categories: secure (trusted) hosts, and insecure (untrusted) hosts. Untrusted hosts are exposed to attacks from malicious users aiming at compromising the functionalities of the programs they run. Two main actors play in this environment: the defender, and the attacker.

The defender is interested in deploying a service or performing a computation that involves the execution of a program \( P \) on an untrusted host (e.g., a Voice over IP service deployed by means of a remote VoIP client such as Skype). The defender deploys his service through a trusted host that has to communicate with \( P \). It is therefore interested in preserving the integrity of \( P \). The defender’s goal can be thus summarized as follows: \( P \) must be used by authorized users for the intended purpose, only. To achieve his goal the defender is free to operate any sort of modification on the program, while preserving its functionality, to make it difficult for a malicious user to tamper with the code. Moreover, it can exploit any functionality provided by the trusted host to perform remote code verification. The proposed model implicitly targets network based applications, i.e., applications that require the network to deploy their functionalities (e.g., client/server applications). Stand-alone programs that, once deployed, can run without communicating on the network, are out of the scope of this paper.

On the other hand, the attacker’s goal is to use a program without authorization or for a purpose different from the one intended by the defender. Whenever the attacker achieves its goal the defender fails. It is worth nothing that, since we are dealing with network applications, the attacker is interested in obtaining the service provided by the defender, and thus he has no interest in performing Denial of Service (DoS) attacks on his platform. It would be meaningless for the attacker to avoid attestation requests from the trusted host or to avoid answering these requests. This would be immediately identified as an attack thus resulting in a cut of the service deployed by the defender.

The assumption in this paper is that a program whose code is genuine works exactly as expected, i.e., it satisfies the defender’s goals. The identification of modifications in the program’s code does not assure that the attacker is actually able to achieve his goal. However, an altered program is the evidence of an anomaly. At a first glance, this assumption may lead to the conclusion that solutions built on top of it would be exposed to attacks aiming at corrupting the program’s computation without actually modifying the executed code. Nevertheless, several software protection techniques do exist that are able to address a wide range of these attacks (see section III). These solutions can still be applied to \( P \) and can be strengthened when code integrity verification is achieved.

The attacker has full control on the untrusted host running \( P \), including its internal hardware, i.e., RAM, cache, processor’s registers, swap space. It can thus read and modify program’s constants, and variables. The attacker can use virtualization, to gain additional control on all hardware aspects visible to the program. Moreover, he has no restrictions on tools and techniques to use to reverse-engineer and tamper with \( P \) (e.g., super-user privileges, debuggers, emulators, tools for efficient comparison and fast search of bit patterns in files, etc.). He also possesses tools enabling to transform programs between different formats and between different levels of abstraction (e.g., disassemblers, decompilers).

The attacker can carry out his malicious activity at any time. This includes off-line attacks completed prior to the execution, and mounted by modifying the program’s executable binary file or its entry point, as well as dynamic attacks mounted at run-time. In the last case, the attacker can use the CPU to investigate and act before and after the execution of any instruction. Moreover, he can interrupt the program based on certain given conditions, including those identifying the best instant to mount the attack.

System libraries and general purpose libraries are controlled by the attacker, along with the operating system. As a consequence, the attacker can use system calls, the input/output subsystem, the network stack, the memory management subsystem, etc. to perform his actions. Network traffic is fully visible and changeable for the attacker. Nevertheless, the attacker has limited computing power, i.e., he cannot easily break public key systems nor perform brute force attacks.

With the assumption of such a powerful attacker, most of state-of-the-art protection techniques will reduce their actual effectiveness, thus motivating the need of introducing additional and stronger protection techniques such as the one proposed in this paper.

III. RELATED WORKS

Several research activities approached the problem of software protection against tampering. Pure software protection techniques can be grouped, at a first instance, into local and remote solutions.

In local approaches, the defender builds tamper proof methods to protect the application directly into the program. Once deployed, the program is under full control of the attacker, and the protection must rely solely on the strength of the implemented techniques, thus limiting the achievable level of security.

Among local techniques, different approaches rely on transforming the program in order to make it unintelligible, and to avoid data extraction (e.g., cryptographic keys) or meaningful transformations. Obfuscation attempts to thwart reverse engineering by making it hard to understand the behavior of a program through static or dynamic analysis. Both Collberg et al. [14], [15], and Wang et al. [16], presented classes of transformations to attempt to confuse analysis of the control flow.
of a program. Theoretical works on white box computation can also be mentioned in this field [17], [18]. Nevertheless, negative theoretical results assure that it is impossible to build perfect obfuscators [19] even for very common classes of programs [20]. In a real scenario, de-obfuscators shortly follow every novel obfuscation technique, thus allowing to transform back the obfuscated program into a form close to the original one, or at least as comprehensible as the original one.

Self-checking, also referred to as integrity checking, represents a group of techniques that try to achieve tamper-proofing by means of tamper-detection to identify modifications of the program, and tamper-response to take actions when the tampering is detected. Tamper-detection is usually based on check-sampling techniques [21], [22], [15]. One of the main drawbacks of these methods, is that they have been proven vulnerable to memory copy attacks [23]. Solutions have been proposed to overcome this attack by including a large number of guards [24], or by compute a fingerprint of the application code on the basis of its actual execution [25], or even by applying self-modifying code techniques [26]. Nevertheless these approaches do not scale properly because the amount of data, and the complexity of the check-sampling computation is hard to manage in software for real applications. Even if strong local tamper-detection techniques can be implemented, dealing with tamper-response is still a challenge [27]. In fact, once an anomaly is discovered the program must be made unusable (e.g., graceful program degradation, probabilistic failures), but the code devoted to reactions can be usually easily identified and defeated.

Some of the drawbacks of local techniques can be overcome by introducing a remote verification entity checking that the code running on an untrusted host is untampered. In general, proposed solutions in this area assume that any program modification or memory copy attack leads to an inevitably increase of the execution time. Time measuring is therefore exploited to detect attacks. Genuinity [28] is a technique that explores the problem of detecting software running into a simulated hardware environment. SWATT is a technique proposed by Seshadri et al. [29] that performs attestation on embedded devices with a simple CPU architecture by using a software verification function built in such a way that any attempt to tamper with it will increase the computation time. Similarly, the Pioneer remote attestation solution [30] compares the time required to run specific functions whose execution time is optimal, with a pre-computed expected execution time. Freshness is provided by a random challenge. The main problem of these approaches is that they rely on a trustworthy information about the actual hardware running the target application. This assumption is very strong and, even if satisfied, is vulnerable to proxy attacks. Moreover, to allow time accuracy, all activities of the system should be stopped during the measurement phase, thus making dynamic verification impossible.

A slightly different approach based on mobile software agents is proposed in TEAS [31]. An arbitrary piece of code, playing the role of an agent to be executed with full permissions on the untrusted host, is sent by a trusted host as a challenge. The agent must send back an attestation in a limited time. Due to the complexity of programs reverse-engineering, it is assumed that the attacker is unable to send a forged attestation in time.

Whenever pure software solutions fail, additional security can be achieved by means of trusted hardware components installed on the target untrusted host. This hardware creates a source of trust in a hostile environment that the defender can exploit to build protections. Several solutions built on top of the Trusted Platform Module (TPM) [32] have been proposed in the literature [4], [33], [34]. Intel also has an initiative for the development of a secure platform [35]. The main drawbacks of these solutions is that hardware components cannot be replaced in case of design errors or security threats. Moreover, secure hardware components such as the TPM are not widely installed on all type of platforms (e.g., embedded devices), and cannot be easily plugged as external components. Recent publications propose the use of reconfigurable hardware to implement software protection techniques [36], [37]. In particular, Gogniat et al. [37] propose a set of interesting references in this field. Nevertheless, these approaches are mainly devoted to the implementation of security primitives into these blocks. To our best knowledge, no solutions try to combine the benefits of software protection based on mobile code with the strength of hardware protection based on reconfigurable hardware.

IV. PROTECTION SCHEMA

MobHaT (Mobile Hardware proTecTion), the software protection mechanism proposed in this paper, is based on the mixed HW/SW architecture proposed in Fig. 1.

Fig. 1. MobHaT: reference architecture for FPGA based remote code integrity verification

The program (P) is the application to be entrusted by the defender. It consists of a binary code executed on the untrusted host to deploy a given functionality (e.g., on-line game, VoIP client, etc.). We consider here native-code applications, only, i.e., programs that directly run on a physical microprocessor. Programs coded into intermediate languages such as the Java bytecode, and executed on a virtual machine, are out of the scope of the proposed approach. As already introduced in section II, MobHaT focuses on network applications. We introduce here the additional requirement of having strong mutual authentication between P and the verification entity (trusted host). This requirement does not introduce strong assumptions. In fact, authentication mechanisms are common elements of network applications.

The untrusted host (U), is the environment where P is executed, e.g., an embedded system such as a cellphone or a PDA. It is assumed under full control of the attacker. It is a microprocessor based system with the only requirement
that it should be provided with a reconfigurable hardware facility with full access to the system memory where \( P \) is stored and executed. The reconfigurable hardware can be either an embedded core integrated into a complex System-on-Chip (SoC), or an external device plugged into the system through a standard socket (e.g., PCI, PCMCIA, etc.). The last solution makes this approach suitable also for systems that are not natively provided with hardware reconfiguration facilities.

The trusted host (TS), is the "verifier" in charge of monitoring the integrity of \( P \). TS is assumed as intrinsically trusted by adopting both hardware and software protections (e.g., firewall, etc.) able to shield the node against external attacks. For this reason, no tampering can happen with it. The attacker has neither access, nor full visibility of \( TS \). He may know what operating system and software it is running, but he should not be able to defeat them. This strongly limits its power since static properties of the code running on \( TS \) cannot be inspected. Nevertheless, the adversary can still inspect dynamic behaviors such as the I/O from \( TS \) by interacting with it in a black-box manner, only. The way \( TS \) performs its verification is by observing a set of attestations received from \( P \). The way attestations are generated can be controlled through the exchange of messages (\( m_c \), and \( m_s \)) with \( P \).

The hardware agent (HA), is a monitoring facility, mapped on the reconfigurable hardware provided with \( U \), in charge of computing attestations of the integrity of \( P \). Attestations are generated by means of a direct access to the host main memory, thus bypassing any need of interacting with \( P \) or with the operating system that might be under the control of the attacker. One of the main characteristics of HA to highlight here, is that it is not a static component, but it is sent by \( TS \) in the form of a mobile hardware agent. This allows to periodically replace this component, thus reducing the ability of the attacker of mounting attacks on a large scale.

Finally, the network (\( N \)), represents any communication mechanism that can be later on mapped on a real infrastructure such as the Internet or a local/corporate network. MobHaT relies on a continuous network access between \( U \) and \( TS \). \( P \) should not be able to execute without exchanging messages with \( TS \). This limits the attacker power by reducing the possibility of performing off-line testing with arbitrary inputs.

Given these basic actors, Fig. 1 shows that direct communications are only possible between \( TS \) and \( P \) through \( N \), and between \( P \) and \( HA \) through the internal system bus of \( U \). The communication between \( TS \) and \( HA \) is therefore mediated by \( P \) that might be under the control of the attacker. This in turn originates the need of establishing a secure communication between \( TS \) and \( HA \).

As a consequence of the trustworthiness of \( TS \), every message \( m_s \) it produces is assumed to be correct. Similarly, any code, or hardware component sent from \( TS \) to \( U \) is assumed to be not malicious at the delivery time.

### A. Security requirements

One of the main problems of using reconfigurable hardware to perform remote software attestation, with respect to other hardware approaches (see section III), is that it must be properly configured prior to perform any function. In fact, a FPGA is not provided with a built-in endorsement key, or a "core root of trust" we can rely on. This section therefore presents a set of security requirements we need to guarantee during the implementation of the architecture of Fig. 1.

\( HA \), that is the component in charge of generating attestations proving the integrity of \( P \), is running on \( U \). It cannot therefore be implicitly trusted. This translates into three basic security requirements:

- **authentication of \( HA \)**: \( TS \) must be able to authenticate the identity of \( HA \);
- **unforgeability of valid attestations**: an adversary (i.e., the attacker, and possibly other parties) must be unable to produce valid attestations from a tampered program;
- **correctness**: each party must efficiently generate and verify all required attestations.

Even if implied by the second requirement, we explicitly mention here that we need to guarantee protection from reply, i.e., an attacker must be unable to store and forward previous valid attestations. Additional requirements such as copy identification, fingerprinting, and channel protection, commonly satisfied in the classes of applications we aim at protecting are here not explicitly considered as they can be implemented as software facilities once the integrity of the program’s code has been properly established.

Additionally, it would be useful, but not mandatory, to have the confidentiality of the FPGA configuration bitstream. As it will be shown in section IV-C, confidentiality can be achieved resorting to functionalities commonly available in modern devices.

Among the factors we can use for authenticating \( HA \), we can only exploit the knowledge factor, i.e., \( HA \) and \( TS \) must share a common secret, or at least \( HA \) should own a secret and a way to prove it without actually revealing it. In the first case, we can use symmetric authentication, while in the second we need asymmetric cryptography. The first method is here preferred over the second one to reduce the computational power required by the reconfigurable hardware, and the drawbacks of implementing random generators to obtain key pairs. This means that the FPGA must be provided with a shared secret, reasonably a symmetric key, to be used for authentication purposes (not necessarily for identification of the single device/user). This originates another requirement: memory curtaining. Sensitive portions of the FPGA (e.g., the memory or where secrets are stored) must not be accessible even by the OS. This is a mandatory requirement since the attacker is assumed to control the OS.

### B. Remote attestation protocol

MobHaT tries to guarantee the integrity of both \( HA \) and the binary code of \( P \), by means of a shared session key \( K_S \) (valid at most \( t \) seconds), a secure one-way function \( H \) (e.g., a cryptographic hash SHA-1 [38]), and a symmetric encryption algorithm \( E \). Since FPGAs can efficiently implement block algorithms, we can assume without loss of generality that \( E \) is a block algorithm (e.g., AES [39]).
Fig. 2 shows the proposed remote attestation protocol. It starts with TS generating a remote attestation request. TS is the only actor allowed to perform this operation; both P, and HA cannot initiate the data exchange. In fact, TS will discard any connection not following the proper message flow. This reduces the vulnerability of TS to DoS attacks.

Sending a remote attestation request implies the generation of a random number R and the usage of a request counter i, both encrypted with the shared session key, i.e., auth = $E_{K_S}(R, i)$. R is used as a nonce to avoid reply attacks against remote attestations, and to avoid that requests can be replied to perform DoS on the FPGA. The minimum suggested length for R is the size of the block of E (e.g., 128 or 256 bits). This makes also very unlikely the repetition of random numbers within the session key validity time $t$. Once the request is ready, TS opens an authenticated channel with P that is listening for connections and sends $req = (R, auth)$. Since the attacker has full control on U, the authenticated channel is effective only up to the untrusted host. The hardware agent monitor (HAM) contained in $P$ is in charge of forwarding the received request to HA.

When HA receives $req$, it uses its copy of the session key to extract the random number, and the sequence counter $(R', i') = D_{K_S}(auth)$. It then verifies that $R' = R$ and, as a reply protection, that $i'$ is greater than the previous received one. It is not necessary to have consecutive indices since TS will try with more than one request in case of missing responses. If the verification fails, HA simply ignores the request without further actions. In this way HA cannot be forced to provide information about $K_S$ by flooding it with fake requests.

If $req$ is valid, HA collects a set of information $p$ proving the integrity of $P$, and a set of information $h$ proving its own integrity. It then calculates the keyed digest of this information including the nonce taken from the request: $ra = H(K_S || p || h || R || K_S)$, where $||$ indicates the concatenation operation. The digest $ra$ is sent to HAM that forwards it to TS, and the channel is immediately closed. As an additional security requirement, and to avoid deadlocks, a remote attestation is considered valid only if it is received within a given time.

$TS$ is able to locally recover the same integrity information provided by HA ($p'$, and $h'$) because it has full knowledge both of $P$ and HA. It calculates $ra' = H(K_S || p' || h' || R || K_S)$ and verifies if $ra' = ra$. If the remote attestation is invalid, TS tries with other two consecutive requests. If both fail, it waits for a specific timeout and sends another request. If even the fourth attestation fails, it invalidates the current key and reacts according to a specific policy (e.g., by disconnecting $U$ from the given service). TS continuously logs all requests, taking track of those that failed. The analysis of log information can be exploited to identify suspicious users.

Different techniques can be used to collect integrity information about $P$. For example, it is possible to simply concatenate all loaded memory pages, to perform a random walk in the set of loaded memory pages, or to apply any other technique using Boolean or arithmetic functions to combine information loaded in memory. These techniques constitute a pool of methods to create different hardware agents (see section V). As HA has full access to the system memory both the code of the application, as well as the data computed at run-time can be exploited to generate these proofs.

It is worth noting that, since the part of the software that manages the interaction between $P$ and HA, i.e., HAM, is actually part of the program itself, and is therefore loaded into the program memory area, while performing attestation of $P$ we implicitly perform also the attestation of HAM.

C. Agents

The implementation of the remote code integrity verification protocol proposed in section IV-B exploits the use of two different types of hardware agents: the key agent, and the monitoring agent that actually performs the remote code integrity verification (HA of Fig. 1).

The key agent is basically used to obtain confidentiality of the bitstream for programming the FPGA, and therefore to obtain confidentiality of the monitoring agents. Confidentiality of the bitstream is implemented by resorting to a set of security facilities commonly available on commercial FPGAs. For instance, all Xilinx Virtex family devices have an on-chip decryption that can be enabled to make the configuration bitstream, and thus the whole logic design, confidential. Old Xilinx devices implemented a standard triple DES (3DES) scheme for securing a bitstream, while newer devices moved to AES encryption. The designer can encrypt the bitstream at the component design time, and the FPGA chip then performs the reverse operation decrypting the incoming bitstream and internally recreating the intended configuration. Virtex-II device family stores the internal decryption keys in a few hundred bits of dedicated RAM, backed up by a small, externally connected battery that can only be written from the outside to replace the key, i.e., no read operation is allowed. This battery backed-up key is the most secure solution since keys are erased if the FPGA is tampered with. Given this hardware feature, the key agent has the main purpose of exchanging a key $K_B$ that will be used to encrypt and decrypt the bitstream of the following monitoring agents, thus reducing the ability of the attacker of analyzing the structure of these components.

Since the monitoring agent cannot rely on asymmetric encryption (section IV-A), we developed an ad hoc key establishment protocol (section IV-D) that, resorting to a empirically hard problem, namely the hardness of reverse engineering FPGA components through their bitstream, allows the secure establishment of the shared session key $K_S$. The key is directly hardwired into HA, and never stored into memory.
elements that can be inspected from the outside. Compared to other problems on which present cryptography relies on, our approach is slightly different. It is not based on mathematical conjectures (e.g., the intractability of integer factorization or the RSA problem [40]) but on empirical considerations about complexity of reverse engineering FPGAs. However, since the strength of this empirical approach cannot be precisely measured, and therefore the attacker still has a certain probability to obtain the key, we will exploit the power of mobile hardware agents to perform rekeying very often. Strengths and weakness of this solution will be analyzed in section VI.

Basically, we have only one distinct key agent and several monitoring agents implementing different techniques to collect the information \( p \) and \( h \) used for remote attestation. To further protect the agents, we also exploit the use of obfuscation techniques to make it difficult for the attacker their comprehension and reverse engineering. These techniques can also be used to create a pool of agent clones to use during the agents replacement process. In this context, each given clone should not provide the attacker with information that can be used to tamper with the next one (uncorrelated agents).

According to Barak et al. [19], an obfuscator \( O \) is an efficient probabilistic "compiler" that takes as input a program \( P \) and produces a new program \( O(P) \) such that (i) \( O(P) \) computes the same function as \( P \) and (ii) anything that can be computed from \( O(P) \) can be also efficiently computed given oracle access to \( P \). As nowadays hardware design is fully supported by the use of high level hardware description languages such as VHDL [41], and Verilog [42], or even by the use of standard programming languages such as C, and C++ [43], obfuscation techniques first designed to protect pure software applications can be moved into the hardware design domain. Specific FPGA based obfuscation techniques such as the one proposed in [44], commercial solutions as the one proposed by Helion Technology [45], as well as adaptation of solutions developed for software [14] can be all applied for hardening reverse engineering attacks. Moreover, the complexity of the obfuscated agent can be additionally increased by including bogus hardware, or by using different routing schemes into the device.

Even if negative theoretical results have been provided about obfuscation in software [19], when considering hardware obfuscation, the capabilities of the attacker are severely restricted by the limited availability of tools for performing static and dynamical analysis of the components mapped into an FPGA, as well as to obtain high level descriptions starting from a configuration bitstream, especially if compared with the software domain. Hardware obfuscation can be therefore considered a valid technique to be applied in our specific problem.

Moreover, given the availability of different obfuscation transformations, Ceccato et al. [46] proved that several thousands of different clones of a given piece of code can be created in a limited amount of time, thus supporting the proposed idea of systematically replacing monitoring agents with new versions that require new work for the attacker to be broken. Given an agent \( A \), we write \( A[K] \) to indicate that a key \( K \) is hardwired in \( A \). Let \( O \) be the concatenation of different software and hardware obfuscation techniques. The final obfuscated agent will be denoted as \( O_i(A[K]) \).

D. Key establishment protocol

The most security sensitive part of MobHaT is the key establishment protocol reported in Fig. 3.

![Fig. 3. The key establishment protocol](image)

In fact, since FPGAs are not provided with a factory certified endorsement key, the key establishment protocol is necessary to agree a shared secret.

When \( P \) is first started, the user authenticates itself to \( TS \). Any strong authentication method is equivalent for our purposes (e.g., a secure SSL/TLS channel to perform password-based authentication, certificate-based asymmetric authentication, etc. [47]). At the end of this authentication \( P \) and \( TS \) establish an authenticated channel. Even in this case, the channel is considered secure up to \( U \). The authentication of the user is performed in \( TS \), for this reason the attacker cannot take advantage from controlling \( U \). Every user is associated with a state information including:

- a bistream symmetric encryption key, \( K_{B,curr} \)
- a shared secret key, \( K_{S,curr} \)
- the request counter \( i \) (as in section IV-B),
- the expiration date of the secrets \( t \), and
- the version of the currently installed agents.

The expiration time is decided by \( TS \) according to security considerations. If the secrets expire, the trusted host starts the key establishment protocol. The case of the first connection of the user is analogous to a standard expiration case. The only difference is that there is no shared secret and the bistream symmetric encryption key is assumed to be the default factory key also known by the attacker.

\( TS \) takes the key agent \( ka \), endows it with a bistream symmetric encryption key \( K_B \), encrypts the corresponding bistream using the last agreed bistream symmetric encryption key \( K_{B,old} \) (i.e., \( E_{K_{B,old}}(O_i(ka[K_B])) \), and send it to \( P \) through the secure channel. \( P \) receives the entire bistream and configures the FPGA through \( HAM \).

\( TS \) then selects a monitoring agent \( ma \), endows it with the secret \( K_S \), and applies a set of obfuscation transformations obtaining \( O_i(ma[K_S]) \). It then encrypts the corresponding bistream using the bistream symmetric encryption key \( K_B \) i.e., \( E_{K_B}(O_i(ma[K_S])) \), and sends it to \( P \) through the secure channel. To reduce network latency, \( TS \) does not expect any notification about the correct installation of the key agent.
before sending the monitoring agent; actually the key agent is required to install the monitoring agent. Nevertheless, it expects an acknowledgement, within a given time, of the termination of the configuration phase, to start the remote attestation protocol and to update the user’s state information.

V. MONITORING AGENTS IMPLEMENTATION, AND EXPERIMENTAL DATA

To demonstrate the concepts presented in this paper, this section proposes a possible hardware architecture for implementing monitoring agents to be used in the protection schema presented in section IV. Fig. 4-A shows the reference architecture of U. It comprises a general purpose processor (GPP), an Input/Output (I/O) subsystem, the main memory subsystem (memory), the peripherals subsystem (peripherals), and finally a FPGA based reconfigurable device. All components are finally connected through a dedicated bus (e.g., a standard PCI bus). The architecture is general enough to fit both embedded devices, as well as personal computers and laptops. The only constraint introduced here is to provide the FPGA with direct access to the memory, i.e., it should be allowed to work as a bus master device.

![Fig. 4. Basic architecture of an untrusted host](image)

Fig. 4-B presents an overview of a generic hardware agent mapped into the FPGA where gray parts can be considered as fixed elements while white blocks are dynamically reconfigured during the agents replacement process. The FPGA contains several security primitives that can be dynamically reconfigured to implement different agent clones. They include the basic blocks required to implement the remote attestation protocol proposed in section IV-B, i.e., an encryption \( (E_{K_S}()) \) and a decryption \( (D_{K_S}()) \) block holding the shared session key \( K_S \), a secure hash block \( H() \), and a key agent block \( (ka[K_S]) \) used during the key establishment protocol (see section IV-D). \( ka[K_S] \) is maintained only for the period required to perform the key update. To increase reverse engineering complexity, the key is always hardwired in each block that requires its use.

A very simple micro-controller (mCON) is embedded in the FPGA together with a local embedded memory block (locMEM) to facilitate the attestation procedure. mCON should only provide basic memory access, mathematic, and comparison instructions (e.g., an old Intel 8080 microprocessor). It can be also exploited to implement the FPGA self-reconfiguration through an Internal Configuration Access Port (ICAP) as the one provided by Xilinx devices. To increase protection against hardware attacks, the content of locMEM can be also encrypted with the shared session key \( K_S \). This only requires to have both the encryption and the decryption primitives inserted into the path between mCON and locMEM.

Two interfaces are finally provided in order to manage the communication on the bus (communication interface of Fig. 4-B), and to manage the communication with HAM (application interface of Fig. 4-B).

The simplest way of computing the proof \( p \) proving the integrity of \( P \) is by collecting the memory pages composing the code of \( P \). The agent, thorough mCON explores the memory of \( U \), and, by navigating the page table managed by the operating system, retrieves the memory regions to analyze. The actual attestation is then computed by applying \( H() \) according to section IV-B. This in turn requires to strongly bound the agent to the specific version of the operating system running on \( U \), and to the specific GPP architecture. To dynamically change the way attestations are computed, pages actually used to compute \( p \) can even be included as part of the request.

In the x86 architecture [48], a two level paging mechanism is adopted. The internal control register CR3 of the CPU points to the first level page table of the current executed process. Combining this address with a Page Directory (PGD) offset the first level page table can be fully explored. For each entry in this table, the second level page table can be then navigated through a Page Table Entry (PTE) offset in order to finally retrieve the actual memory pages. This mechanism is depicted in the figure 5. Similarly the x86-64 [48], [49], the recent Intel Atom [50] architecture, and ARM compliant [51] architectures adopt a multilevel memory paging systems, while the PowerPC [52] architecture uses a single level hash table.

![Fig. 5. Hardware monitor architecture](image)

The main limitation of this architecture is the need of forwarding the position of the paging table (e.g., the content of the CR3 register in a x86 architecture) from GPP to the FPGA. If no specific hardware mechanism is implemented to perform this operation in a secure way, i.e., we are dealing with a commercial-of-the-shelf device not specifically designed to guarantee software protection, this operation should be performed by HAM, thus exposing the agent to the memory copy attack. The memory copy attack consists of a fake paging table provided to the agent in order to force it to create the attestation on a memory area containing a genuine copy of \( P \), while a tampered running copy is placed in a different area. However, software solutions have been proposed to tackle this...
attack. Data computed at run time can be included in \( p \) to guarantee that the memory area used to compute the attestation is actually executed [24], [25]. Alternatively the capabilities of \( HA \) can be exploited to dynamically modify at run-time the code of \( P \), thus guaranteeing that only executing the code area accessed by \( HA \) the correct attestation can be computed [26].

A. Experimental data

As a proof-of-concept we implemented a prototype of the proposed protection schema including a hardware agent reflecting the architecture of Fig. 4.

The implementation includes the following design choices:

- an untrusted host built into a Intel x86 based desktop computer running the linux operating system with a 2.6 kernel, provided with a plugged PCI board equipped with a Xilinx Virtex-II FPGA core;
- a trusted server implemented into a dual Intel Xeon quad core also running the linux operating system, connected with the untrusted host through a local area network.

A simple implementation of \( TS \) including a repository of three monitoring agents, and key agents including three different keys as been developed to simulate hardware agents replacement into the system. SHA-1 [38] has been selected to implement \( H \), while AES [39] has been implemented to perform encryption and decryption of attestations.

The method has been used to secure a simple program able to search for anagrams of a string. It is a network application composed of a trusted part running on a trusted host requiring the computation of anagrams, and an untrusted part, running on an untrusted host, able to perform this computation. Whenever the trusted part requires the computation of an anagram, it contacts the untrusted host by issuing a request including the string for which to calculate the anagrams and the name of a target dictionary. The untrusted part performs the computation and then returns the list of identified anagrams. The communication is performed over a TCP connection.

Table I gives some quantitative results that can be used to determine the performance of our experimental setup. The numbers given in the table are representative of a large range of applications that can be deployed on the architecture described above.

By considering the proposed experimental data, considering a service provider with a 1 Gbps network link, and about 3 MB to replace two agents, about 3.6M replacements/day can be managed.

VI. SECURITY ANALYSIS

This section concludes the presentation of MobHaT by proposing a security analysis highlighting strengths, and especially weaknesses to address in future research activities.

If \( H \) and \( E \) are secure, and \( K_S \) is secret, proving that the remote attestation protocol of section IV-B guarantees integrity and authenticity of the attestation request from \( TS \), the integrity of \( P \), and the integrity and authenticity of the response from \( HA \) is straightforward. In fact, \( TS \) performs a unilateral symmetric authentication using random numbers, and the FPGA authenticates itself by performing a unilateral authentication using a keyed one-way function [40]. The integrity of \( P \) is also assured by using the same one-way function.

Since \( TS \) is unique (every program can be contacted only by it), and the answer from \( HA \) should arrive in a limited amount of time inside the same authenticated channel, there is no need to insert identifiers to prevent reflection attacks. \( R \) is also sent in clear due to the loose verification of sequence numbers (simply \( i_{curr} > i_{prev} \)).

Opening an authenticated channel between \( U \) and \( TS \) is a computationally expensive task, even if techniques based on resume functionalities may strongly reduce its impact (e.g., SSL/TLS session ID renewed before starting the key establishment protocol, and having the same validity period of the secrets). Nevertheless, it is a required price to avoid DoS and man in the middle attacks. The trusted host could be flooded with several randomly generated strings of bits of the same size of the digest sent by \( U \), and \( U \) may be flooded with fake requests.

We considered here for the first time Man in the Middle (MITM) attacks. Even if out of the scope of this paper, since strictly connected to the specific functionalities deployed in \( P \), they should be considered when looking at the implementation of a more general framework for remote code integrity verification. The MITM controls the network and can perform passive attacks (eavesdropping) as well as active attacks (packet modifications), and may be interested in performing DoS attacks. The goal of the MITM is different from the attacker’s goals presented in section II. He has no personal advantage in allowing the use of tampered programs but he is not explicitly against it. In this context, the attacker (as defined in section II) is actually an allied in countering the attacks from the MITM. In fact, he needs the service provided by \( TS \) to attest the integrity of \( P \). The attacker will actually protect the untrusted host and will not allow third parties to access or make unavailable the FPGA.

Since \( HA \) is designed using state-of-the-art algorithms, e.g., AES and SHA-1, the overall security of the remote attestation schema depends on \( K_S \). The security analysis must therefore concentrate on proving that the key establishment protocol is secure.

The correctness of the protocol is guaranteed since it involves techniques that are computationally feasible (section IV-C). It is computationally easy for \( TS \) to prepare an agent with an embedded key.

Concerning the soundness, we must prove that \( TS \) and \( HA \) share the same key \( (K_S) \), and it is not feasible for any party but \( TS \) and \( HA \) to obtain that key. It follows immediately that when the protocol correctly terminates the FPGA is configured with the right monitoring agent \( ma \). Therefore, \( TS \) and \( HA \) share the same key because \( K_S \) is embedded in \( ma \). In fact, the attacker has no interest in configuring the FPGA with wrong agents and the MITM cannot force the configuration of wrong agents because they are sent to \( U \) inside an authenticated channel. If the MITM has full control over the network, it may block the messages (requests, attestations, agents). Nevertheless, this vulnerability is a common threat with network applications.
Once proved the correctness and the soundness of the protocol, it is necessary to prove that it is not feasible for the attacker to recover $K_S$ by reverse engineering the monitoring agent. Proving this property is a non-trivial task. In fact, in modern cryptography, the notion of infeasibility is strictly related to the notion of computation. In particular, anything beyond polynomial-time computations is considered not feasible (e.g., RSA and DH problems \[53\].) However, it is not possible to analyze reverse engineering complexity only by means of computations. The human factor becomes predominant. Since the evaluation of infeasibility is here mainly empirical, we introduced additional (empirical) techniques to strengthen our solution including the usage of the key agent, and periodic key replacement by means of uncorrelated monitoring agents clones. Actually, the key agent does not take an active role during the attestation protocol. However, it provides a significant contribution in achieving a high level of empirical protection. Indeed, to derive $K_B$ an attacker must first reverse engineer the key agent in order to extract $K_B$. Only when $K_B$ is extracted he may obtain the bitstream of the monitoring agent, and by reverse engineering also this component he may try to deduce $K_S$. To control and limit the time the attacker has to reverse engineer agents, the defender may also operate on the expiration time $t$. This parameter can be regulated according to the security requirements, but it is reasonable to consider multiple of days or even weeks. By deciding to replace both monitoring agents and key agents, before reverse engineering a monitoring agent to extract the current $K_S$, the attacker is forced to comprehend all previously received key agents. Due to the hypothesis that clones are uncorrelated, the time to mount an attack increases linearly with the number of replacements. Furthermore, the agent replacement guarantees that even if the attacker achieved his goals, it should start from scratch with a completely new version of the agents. We define these “time-limited successful attacks”.

Using hardware agents replacement also provides protection against massive attacks. Using a single key agent, and a single monitoring agent, exposes programs to potential mass deployment of attacks. In fact, an attacker could distribute to a large audience a tampered version of the program able to circumvent our protection technique. By reverse engineering the agent it would be possible to write a “crack” obtaining $K_S$ and emulating the behavior of our hardware agent. Since thousands of equivalent agents can be produced, the attacker should reverse engineer most of them to have an (almost) effective crack. Due to the hypothesis that clones are uncorrelated, the time to reverse engineer all of them is counted in years. Bigger than usual life-time of a program.

Finally, the establishment protocol is not vulnerable to know key attacks, i.e., compromising any past key $K_S$ does not allow the attacker to compromise future session keys that are generated by $TS$, independently.

We assume that the applications the defender is deploying are not intended for high security domains (e.g., military, government). For this reason, the attacker is not motivated in performing invasive hardware attacks to break the secret contained in the FPGA (e.g., by means of electron microscopes). These type of attacks would be too expensive if compared with the advantages that may derive from a misuse of the application.

We can therefore conclude that, if the given requirements on the program $P$ to protect are satisfied, unless very efficient and automated methods to reverse engineer FPGA are developed, this approach is secure against an attacker as in section II and against MITM attacks.

### VII. Conclusion

This paper proposed the use of mobile hardware agents implementing software monitoring techniques for securely producing and delivering attestations of code integrity of an application. A protection framework including both protocols design and hardware architectures has been presented. One of the main contributions of the paper is the use of mobility of hardware components to make it difficult for an attacker to reverse engineering hardware monitors thus defeating the protection technique. Experimental data on scalability issues give promising results.

Future activities include the development of a complete implementation of the framework able to consider the entire lifecycle of the program including the development of management workflows, the standardization, and optimization of all administrative procedures. This will also include the design of an agents’ factory (agency), the reduction of the network overhead through a distributed set of trusted hosts, and the definition of a standard set of reactions to tampering detection. The production of agents, is essentially an off-line process. Nevertheless, after the generation, they need to be catalogued and associated with the secrets they contain in order to be quickly available to $TS$ when needed. In a large scale scenario, the same agent may be sent to different users with different keys. Also distribution should be designed carefully to reduce coalitions. Finally different tamper reaction procedures can be designed in order to better fit the requirements of different classes of applications.

### REFERENCES


Distributing Trust Verification to Increase Application Performance

Mariano Ceccato\(^1\), Jasvir Nagra\(^2\), Paolo Tonella\(^1\)
\(^1\) Fondazione Bruno Kessler—IRST, Trento, Italy
\(^2\) University of Trento, Italy
\{ceccato, tonella\}@fbk.eu, jas@dit.unitn.it

Abstract

The remote trust problem aims to address the issue of verifying the execution of a program running on an untrusted host which communicates regularly with a trusted server. One proposed solution to this problem relies on a centralized scheme using assertions and replication to withhold usable services from a tampered client. We show how to extend such a scheme to a distributed trusted hardware such as tamper-resistant smartcards. We compared the performance and security of the proposed distributed system to the original centralized scheme on a case study. Our results indicate that, compared to a centralized scheme, our distributed trust scheme has dramatically lower network traffic, and smaller memory and computational requirements on the trusted server.

1 Introduction

There are two key trust issues that arise in an open distributed computing setting. Firstly, a client that joins a distributed system may wish to be sure that the code it receives to execute has not been tampered and when executed it is not malicious. Secondly, a server would want to be sure that any computation it requests from a client is performed according to the conditions and rules prescribed by the server. The server is not willing to provide its services to untrusted clients, which may behave maliciously or unfairly with respect to the other clients.

While cryptographic signature schemes and sandboxed clients largely address the first of these concerns, the latter concern— that of software integrity—remains an open problem. The Remote trusting problem is a particular instance of the software integrity problem in which a trusted host (server) wishes to verify that an untrusted host (client) is executing according to its expectations at the point when the client requests a service.

A solution to the remote trusting problem is based on moving the tamper-sensitive part of the client computation to the server. Such a solution was investigated by Zhang and Gupta [13], in the context of protection from illegal software copying. Copy-sensitive parts of the clients are sliced and moved to the server so as to make copying ineffective, if based on the client code only. Another solution, investigated in [2], focuses on the notion of invalid state and exploits the barrier slicing technique to move the invalid-sensitive part of the client state to the server. Both solutions are based on the assumption that the core of trust is in the server and that every computation moved to the server becomes intrinsically safe.

The main problem of these two solutions based on slicing is that the server, which is the only reliable source of trust, is overloaded with computations that cannot be performed safely on the client. When the server is accessed by a high number of clients concurrently, it might be impossible to ensure an adequate quality of service.

In this paper, we propose a distributed trust architecture, which takes advantage of trusted hardware, such as smartcards, residing on each client. The core of trust is split between central trusted server and local trusted hardware, so as to delegate to the local trusted hardware everything that does not need a centralized service. The server keeps only its original services, which by design cannot be distributed, so that its load is unchanged, compared to the original application. Additional trust is achieved by means of the local trusted computations performed by the smartcards. The slices of trust-sensitive code are moved to the local trusted hardware.

2 Background

In this section we summarize the remote entrusting problem and the centralized solution. More details are available in the previous paper [2].
2.1 Remote trust verification

Remote trust verification involves a trusted host (server) $S$, an untrusted host (client) $C$ and a communication channel between the two. The integrity of the application $P$ running on $C$ has to be verified whenever a communication act occurs between $S$ and $C$.

We assume a scenario in which the application $P$ requires a service delivered by $S$. To receive this service a communication channel is established between $C$ and $S$ and some messages are exchanged:

$$C[s] \xrightarrow{m} S \quad \text{and} \quad S \xrightarrow{v} C[s]$$

where $s$ is the current state of application $P$ running on $C$ and $m$ is a message that requests some service from $S$. Once $S$ receives the request $m$ it replies by sending the message (service) $v$.

The state $s$ of the client application during a communication with $S$ is a valid state when it satisfies certain validity properties expressed through an assertion $(A(s) = \text{true})$. In order for $S$ to trust the application $P$ upon the execution of a communication act, $P$ has to exhibit a valid state. The only way in which $S$ can verify the validity of the application $P$ is by analyzing the message $m$ that $C$ has sent. $S$ trusts $P$ upon execution of the communication act $C[s] \xrightarrow{m} S$ if $E(m) = \text{true}$, where $E$ is an assertion checked by the server $S$.

Thus, the remote trusting problem consists of finding a protection scheme such that verifying $E(m)$ is equivalent to having a valid state (i.e., $E(m) \Leftrightarrow A(s)$).

A protection mechanism is not sound (attacker wins) whenever the server is trusting the client, but the current state of the client is not valid ($E(m) = \text{true} \land A(s) = \text{false}$). A protection mechanism is not complete when the server does not trust a client that should be trusted ($E(m) = \text{false} \land A(s) = \text{true}$).

In the attempt to break a given protection mechanism, we make the assumption that the attacker can: (1) reverse engineer and modify the code of $P$; (2) alter the running environment of $P$, for example through emulators or debuggers, and dynamically change the state of $P$; (3) produce static copies of $P$ and execute multiple copies of $P$ in parallel, some of which are possibly modified; and, (4) intercept and replace any network messages upon any communication act.

2.2 Using barrier slicing for remote trust

When the server $S$ delivers a service $v$, the client $C$ can make some use of $v$ only if its state is consistent with the state in which the service was requested. More specifically, we can make the assumption that a (possibly empty) portion of the state $s$ of $C$ must be valid in order for the service $v$ to be usable by the client. We call this substate of the client the \textit{safe} substate. Formally, such substate $s_{\text{safe}}$ is the projection of the state $s$ on the safe variables of the client. Intuitively, when the service $v$ is received in an invalid substate $s_{\text{unsafe}}$, the application cannot continue its execution, in that something bad is going to happen (e.g., the computation diverges or blocks). The complement of the safe substate is called \textit{unsafe} substate and is formally the projection of the state on the unsafe variables: $s_{\text{unsafe}}$.

The intuitive idea behind the usage of barrier slicing for remote trusting is to move the portion (slice) of the application $P$ that maintains the variables in $s_{\text{unsafe}}$ to the server, in order to prevent the attacker from tampering with them. To limit the portion of code that needs to be moved, \textit{barrier slicing} instead of regular slicing is used.

The regular, backward slice for a variable at a given program point (slicing criterion) gives the subprogram which computes the same value of that variable at the selected program point as the original program. Hence, slicing on the unsafe variables at the communication acts gives the portion of client code that, once moved to the server, ensures correct and reliable computation of $s_{\text{unsafe}}$. Since it is the server which computes $s_{\text{unsafe}}$, by executing one slice per client, no tampering can occur at all (the server is assumed to be completely trusted).

We can notice that the computation of the moved slices on the server is somehow redundant. In fact, such computation may involve values from $s_{\text{safe}}$ which are ensured to be valid and thus do not need be recomputed on the server. In other words, whenever the computation carried out in a slice moved to the server involves a safe variable, it is possible to use the value obtained from the client (extracted from $m$), without any need to recompute it. The technique to achieve such a slice portion is called \textit{barrier slicing} [4, 5], with the barriers represented by the safe variables which do not need to be recomputed. Technically, when including the dependencies in the backward transitive closure, the computation of the slice is stopped whenever a safe variable is reached.

3 Distributed trust verification

3.1 Centralized trust architecture

The barrier slicing solution to the remote trusting problem is based on a centralized architecture. The slice is moved onto the server and instructions are added to the client to cause the server execute the slice when needed. Data required to execute the slice is sent to the server by the client and required values are requested by the client when they are needed. This involves additional communication and synchronization, and requires the execution of a stateful computation (the moved slice) for each currently running client.
The additional communication and synchronization messages required in this scheme add a lot of network overhead to the application. Several optimizations were outlined in the original proposal [2] to reduce the number of communication messages required. In spite of these optimizations, in some applications the slowdown introduced by these additional communication acts may result in an unacceptably large slow down of the application.

The second performance penalty depends on the amount of computation that must be performed by the server. If the server is supposed to serve a high number of clients, the centralized solution [2] may not scale, overloading the trusted server with a large number of concurrent slice executions, which demand too much of computing power or memory. In fact, slice execution must be instantiated on a per-client basis and may involve substantial memory, since it is by definition a stateful computation.

### 3.2 Distributed trust architecture

In a distributed trust scheme, each client has a tamper-resistant computational device (e.g., smartcard) which is trusted by the server. In this work, we investigate a distributed trust architecture, used in conjunction with barrier slicing in order to achieve trusted computation of $s_{|\n_s}$ with minimal performance penalties. We make the assumption that the client is connected to some trusted, programmable hardware, such as a smartcard, possibly attached to the USB port. This hardware usually has limited computing and memory power. Hence, it is not possible to move the entire client application to it and solve the remote trusting problem in this way. However, the computing capabilities of such a device are probably sufficient to execute a barrier slice of the client, thus ensuring trusted computation of $s_{|\n_s}$.

In the rest of the paper we will use the term smartcard to generally refer to any tamper-resistant hardware. The same protection mechanism can be implemented using any secure device, such as secure coprocessors, USB dongles or secure mobile devices.

Intuitively, the barrier slices that ensure remote trusting are moved and executed on the smartcard. The client continues to run the portion of application that is non-sensitive, from the security point of view (e.g., the GUI), or that is intrinsically secured (i.e., computation of $s_{|\n_s}$). The client continues to communicate with the server for any service $v$ necessary to continue its execution. A virtual, secure channel is established between the smartcard and the trusted host, so that whenever the smartcard detects a deviation from the valid states, the server is notified and stops delivering its services to that particular client.

The program on client $C$ in a state $s$ sends a message $m = f(s)$ to the smartcard. Depending on the power of the smartcard, the smartcard either encrypts $m$ using a key or verifies the assertion and encrypts a message to indicate the validity of $m$. The smartcard sends this encoded message $Enc(m)$ either directly or via the client to the server. These encoded messages serve as a secure virtual channel between the smartcard and the server. Based on this message, the server sends the client a service $v$. The assertion $E(m)$, checked either on the server or on the smartcard, determines whether the server should trust the client or not. The load on less powerful smartcards can be further reduced by allowing the client to send unencrypted messages $m$ directly to the server. Instead, the smartcard periodically sends a checksum of recent messages to the server allowing the server to test the veracity of messages it has received since the last checksum, allowing it to detect tampered messages and withhold further service. In this scheme, there is a trade-off between the power required in the tamper-resistant hardware and length of time an attacker can successfully receive a tampered service before being detected by the server.

The architecture described herein addresses the two main performance penalties for the application running on the client. The network overhead is restored to levels comparable to the initial application. All communication acts necessary for the client to obtain the values of variables in $s_{|\n_s}$ do not transit over the network any longer. They go directly to the local smartcard. The computation and memory overhead are eliminated, since the barrier slices are not executed on the server any more. It is the smartcard that contains their code and runs them locally. Compared to the initial, untrusted architecture, the client and server pay minor penalties related to the authentication messages that are generated by the smartcard and transit over the network.

### 4 Program Transformation

As in the case of the centralized protection, the variables that need to be protected must be manually chosen by the user. Once these variables have been selected, the code to be moved to the smartcard can be automatically computed and the client code can be transformed. The transformed code is eventually installed on the client. In this section we cope only with code modifications that are required to apply the distributed protection mechanism. Other changes, required to make the code run on special hardware are out of the scope in this paper. Appropriate cross compilers or manufacturer proprietary tools should be used.

The transformation steps are described with reference to the example in Figure 1. Let us consider the fragment of code in Figure 1(a). If we consider the $n$-th communication, the barrier $B_n$ is given by instruction 1, while the slicing criterion $C_n$ is given by instructions 10 and 12. By computing
1 \[ x = x \cdot a; \]
2 \[ a = a + x; \] send\(b(m_i); \) receive\(b(k_h); \)
3 \[ a = x + a; \]
4 \[ x = x + 1; \]
5 \[ \text{while (c) \{ } \]
6 \[ a = a + x; \]
7 \[ x = x + a; \]
8 \[ \text{if (c) \{ } \]
9 \[ a = 2 \* x; \]
10 \[ \text{else \{ } a = a + x; \}
11 \[ x = x + a; \]
12 \[ \text{end while; } \]
13 \[ \text{end if; } \]
14 \[ \text{send\(b(m_i); \)} \]
15 \[ \text{receive\(b(k_h); \)} \]

C1 \[ x = x \cdot a; \]
C2 \[ \text{sync(); } \]
C3 \[ \text{sync(); } \]
C4 \[ x = x + 1; \]
C5 \[ \text{while (c) \{ } \]
C6 \[ sync(); \]
C7 \[ x = x \cdot \text{ask\("a"\); } \]
C8 \[ x = x \cdot \text{ask\("a"\); } \]
C9 \[ \text{if (c) } \]
C10 \[ \text{then \{ sync(); } \]
C11 \[ x = x \cdot \text{ask\("a"\); } \]
C12 \[ \text{else \{ } \]
C13 \[ x = x + \text{2\*ask\("a"\); } \]
C14 \[ x = x \cdot \text{ask\("a"\); } \]
C15 \[ \text{send\(b(m_h); \) receive\(b(k_h); \)} \]

S1 \[ a = a + x; \]
S2 \[ \text{sync(); } \]
S3 \[ x = m; \]
S4 \[ \text{if A(x, a) then } \]
S5 \[ \text{sendAuthenticityTag\(m_h); \text{ else } sendTamperedTag(); } \]
S6 \[ a = x + a; \]
S7 \[ \text{sync(); } \]
S8 \[ x = x + 1; \]
S9 \[ \text{while (c) \{ } \]
S10 \[ a = a + x; \]
S11 \[ \text{sync(); } \]
S12 \[ \text{sync(); } \]
S13 \[ x = x + a; \]
S14 \[ \text{send\(n(m_k); \)} \]
S15 \[ \text{receive\(n(k_n); \)} \]

Figure 1. An example of the proposed protection scheme: (a) original client, (b) modified client and (c) corresponding smartcard.

the barrier slice, we obtain:

\[ \text{Slice}_1(C_n, B_n) = \{12, 10, 9, 8, 7, 6, 5, 4, 3, 2\} \]

4.1 Client code

The transformation of the client consists of removing some of the statements in the barrier slice and introducing some extra communication with the smartcard, to retrieve the needed values. The transformation is composed of the following steps:

- Every message \(m\) sent to the server is also sent to the smartcard.

- Every unsafe variable definition in the slice (Figure 1(a) statements 2, 3, 6, 10 and 12) is replaced by the instruction \textit{sync()} (Figure 1(b) statements C2, C3, C6, C10 and C12). This message corresponds to a synchronous blocking communication, which means that the client has to wait for the answer from the smartcard. The smartcard sends an \textit{ack} only when its execution reaches the corresponding \textit{sync()} (Figure 1(c) statements S2, S8, S12, S17 and S19).

- Every use of variable \(a \in \text{Unsafe}\) on the client is replaced by an \textit{ask\("a"\)} that requests the current value of \(a\) from the smartcard (Figure 1(b) statements 7, 8, 11, 13 and 14).

- The last change involves input values (i.e., user input, file read), which must be forwarded to the smartcard as soon as they are collected by the client application.

- On the client a new process is added (not shown in Figure 1(b)), that just waits for encrypted messages coming from the smartcard and forwards them as they are to the server. In this way a virtual secure channel is established between the smartcard and the server.

4.2 Smartcard code

The code to be uploaded onto the smartcard aims at: (1) making the smartcard able to run the slice computing the \textit{Unsafe} variables; (2) keeping it synchronized with the client; and, (3) verifying the validity of the \textit{Safe} variables. The transformation of the client is composed of these steps:

- The very first change to apply is to copy the barrier slice \(\text{Slice}_1(C_n, B_n)\) to the smartcard. The smartcard has to boot strap the slice as the original client does (e.g., data structures must be initialized).

- The slice is fed with any input coming from the client.

- As soon as the smartcard receives a message \(m\) from the client, the validity of the client’s state is verified (statements S4–S6, S21–S23), after extracting the values for the \textit{Safe} variables (statements S3, S20). If the state is considered valid an \textit{authenticity tag} is sent...
to the server through the virtual secure channel. Otherwise, the server is notified about the identified attack, through a tampered tag. Tags are encrypted and signed using a secret key hidden in the smartcard, thus they are not visible to the attacker. They include the message received from the client.

- Whenever a sync() statement is reached, the current values of the Unsafe variables are saved, after synchronizing with the client.

- A smartcard process (not shown in Figure 1(c)) replies to each client’s ask() by sending the currently saved value for the requested variable.

Figure 1(c) shows the code running on the smartcard after the transformation. Instruction 2 of the original application contains a definition of variable $a \in \text{Unsafe}$. In the client, this instruction is replaced by a sync() (instruction C2), corresponding to the smartcard’s sync() S2. Upon synchronization, when the client’s execution is at C2 and the smartcard’s execution is at S2, the current value of the unsafe variable $a$ is saved on the smartcard. The smartcard can then proceed until the next sync() (instruction S8), and any ask() issued by the client is replied by a parallel smartcard process sending the stored value of $a$ (i.e., the value produced at S1). We can observe that instructions 11 and 13 are not duplicated on the smartcard, since they do not belong to the considered barrier slice.

4.3 Server code

In contrast to the centralized solution, the code of the server (not shown in Figure 1) is affected by only minor changes. A new process is added that establishes the virtual secure channel with the smartcard and waits for authenticity tags. In case they are not received at the expected rate, an invalid or a tampered tag is received, or the message $m$ brought by the tag does not correspond to the message $m$ received from the client, this process notifies the original server to stop delivering the service to the corresponding client. In alternate implementations where the assertion $E(m)$ is computationally expensive and the resources on the smartcard are meager, code to test the assertion is added to the server.

5 Experimental results

Both the distributed and centralized protection architectures have been applied on the same case study, a network application, and a scalability assessment has been performed on them.

5.1 Case Study

CarRace is a network game, the client of which consists of around 900 lines of Java code. The application allows players to connect to a central game server and race cars against each other. During the race, each player periodically sends data about the car position and direction to the server, which then broadcasts the data to the other clients allowing them to render the game on their screen. The fuel is constantly consumed, and a player must periodically stop the car and spend time refueling.

There are many ways a malicious user (the attacker) can tamper with this application and gain an unfair advantage over his competitors. For example, he can increase the speed over the permitted threshold, change the number of performed laps or avoid refueling by manipulating the fuel level. Unfortunately not all the variables that must be protected against attack are in Safe. The attacker cannot tamper with the position (variables $x$ and $y$), because the displayed participants’ positions are those broadcast by the server, not those available locally. The server can check the conformance of the position updates with the game rules (e.g., maximum speed). The other sensitive variables of the game (e.g., gas) are Unsafe and must be protected by some extra mechanism, such as barrier slicing. A barrier slice has been computed using the Unsafe variables as criteria and Safe variables as barrier. The barrier slice is quite small, just 120 lines of code (14% of the entire application) so it can fit low cost, commercially available, secure hardware, such as smartcards.

5.2 Scalability Analysis

The two different protection mechanisms (centralized and distributed architecture) have been applied to the case study application. In order to analyze how they scale when the number of connected clients increases, the server performance has been recorded in terms of memory consumption, number of threads and network traffic (i.e., exchanged messages). The server has been run respectively with 2, 4, 6, 8, 10 and 12 clients (the race is played pairwise). From the trend of the plotted data, the scalability of the system was assessed. In order to make the measurement objective and repeatable, a “softbot” was developed, able to play the race by driving the car through a sequence of check points.

Figure 2 shows the amount of memory (bytes) allocated on the heap of the Java Virtual Machine that runs the server. In the case of two clients, the memory required by the centralized solution is about the double of the distributed solution memory size (988 Vs 524 bytes). The slope of the line that connects the experimental points suggests the scalability of the two solutions. By linear fit, we can determine that while the centralized protection requires to allocate about
220 bytes per each new client, the distributed one requires just 32 bytes per client (15%).

Results indicate that in the CarRace application, the slice executing on the smartcard consumed 820 bytes of heap memory compared to 2325 bytes on the original client. In other words, less than 40% of the memory in use in the original client ends up being required on the smartcard. Furthermore, the slice required less than 25% of the CPU cycles of the original application. While these are significant computational resources, they are a considerable improvement over using tamper-resistant hardware to protect the entire client application. The resources required to execute the smartcard slice vary greatly from application to application, so we expect higher or lower benefits depending on the given case. Optimizations are also possible for specific applications (see previous sections).

If we consider the threads running on the server, we can see on Figure 3 that the distributed solution is less demanding when the number of clients increases. In fact, while the centralized solution requires 4 new threads per served client, the distributed approach requires just one thread per client (25%). In fact, all the threads in the barrier slice must be replicated on the centralized server, for each connected client, while in the distributed solution just one more thread is necessary, to handle the encrypted network communication.

Figure 4 shows how network traffic (number of exchanged messages) changes on the server when the number of clients increases. In the distributed solution the number of messages exchanged by the server is the same as the original (i.e., non-protected) application, because only the original messages go through the server (145 messages per client). In fact, all the messages that keep the unsafe state updated go through the secure hardware (e.g., smartcard). In the centralized solution the server has to handle either the original messages and the security-related messages, causing the network traffic to increase faster (1743 messages per client, according to a linear fit). In the distributed solution, the server network traffic growth is about 8% of the centralized solution.

6 Related works

The problem of remote attestation of software has a colorful and long history. The key idea of a “trusted computing base” (TCB) can be traced to the Orange Book [7] and Lampson [6]. Lampson defines the TCB as a “small amount of software and hardware that security depends on”. In this context, security was assured by the TCB because the operating system and hardware were assumed to be known, trusted and inviolable. More recently, trusted hardware schemes for remote attestation have been proposed. The Trusted Computing Group [8] and Microsoft’s Palladium [1] have proposed several schemes based on a secured co-processor. These devices use physical defenses against tampering. The co-processor stores integrity measurement values that are later checked externally. The increased cost of manufacture and prohibitive loss of processing power due to the required cryptography has largely limited the mainstream adoption of these solutions.

Alternatives to custom trusted hardware are represented by software-only solutions that rely on known hardware. Swatt [10] and Pioneer [9] apply respectively to embedded devices and desktop computer. At run-time they compute a checksum of the in-memory program image to verify that no malicious modifications has occurred. They take advantage of an accurate knowledge of the client hardware and memory layout to precisely predict how long the checksum computation should take. Detection of tampering is based on an
observed execution time that exceeds the upper bound, under the assumption that any attack introduces some levels of indirection, which increases the execution time.

Detailed client hardware knowledge is a reasonable assumption when a collaborative user is interested in protecting her/himself against malware. It does not apply to malicious users who are willing to tamper with the hardware and software configuration or provide incorrect information about it.

If checksum computation time can not be accurately predicted, the memory copy attack [12] can be implemented to circumvent verifications. A copy of the original program is kept by the malicious user. Authenticity verification retrieves the code to be checked in data mode, i.e., by means of proper procedures (get code) that return the program's code as if it were a program's datum. In any case, the accesses to the code in execution mode (i.e., control transfers to a given code segment, such as method calls) are easily distinguished from the accesses in data mode. Hence, the attacker can easily redirect every access in execution mode to the tampered code and every access in data mode to the original code, paying just a very small performance overhead.

Kennon and Jamieson [3] propose a scheme called Genuinity which addresses this shortcoming of checksum-based protections by integrating the test for the “genuineness” of the hardware of the remote machine with the test for the integrity of the software that is being executed. Their scheme addresses the redirection problem outlined above by incorporating the side-effects of the instructions executed during the checksum procedure itself into computed checksum. The authors suggest that the attackers only remaining option, simulation, cannot be carried out sufficiently quickly to remain undetected. Shankar et al. [11] propose two substitution attacks against Genuinity which exploit the ability of an attacker to add code to an unused portion of a code page without any additional irreversible side-effects.

7 Conclusions

In this paper we address the remote trusting problem, verifying the healthy execution of a given application that runs on an untrusted host. Our solution consists of a distributed architecture. The application to protect is divided into two segments using barrier slicing. The portion that keeps sensitive variables up to date is moved to local, secure hardware, in order to protect it against tampering.

This solution represents an improvement of the previous centralized solution, proposed in [2], where the sensitive application part is moved to the server. On a small case study, we observed that the centralized architecture causes unacceptable server overhead, when many clients are connected. The distributed solution has better scalability. It requires considerable less server resources in terms of allocated memory (15%), threads (25%) and network traffic (8%), while providing the same level of protection. On the other hand, the portion of code to move onto the local, secure hardware (14% of the total application) is small enough to fit a smartcard.

References

Smart Cards and remote entrusting

Jean-Daniel Aussel, Jerome d’Annoville, Laurent Castillo, Stephane Durand, Thierry Fabre, Karen Lu, Asad Ali

Gemalto
Technology & Innovation
{jean-daniel.aussel | jerome-d-annoville | laurent.castillo}@gemalto.com

Abstract

Smart cards are widely used to provide security in end-to-end communication involving servers and a variety of terminals, including mobile handsets or payment terminals. Sometimes, end-to-end server to smart card security is not applicable, and smart cards must communicate directly with an application executing on a terminal, like a personal computer, without communicating with a server. In this case, the smart card must somehow trust the terminal application before performing some secure operation it was designed for. This paper presents a novel method to remotely trust a terminal application from the smart card. For terminals such as personal computers, this method is based on an advanced secure device connected through the USB and consisting of a smart card bundled with flash memory. This device, or USB dongle, can be used in the context of remote untrusting to secure portable applications conveyed in the dongle flash memory. White-box cryptography is used to set the secure channel and a mechanism based on thumbprint is described to provide external authentication when session keys need to be renewed. Although not as secure as end-to-end server to smart card security, remote entrusting with smart cards is easy to deploy for mass-market applications and can provide a reasonable level of security. ¹

1 Introduction

Smart cards are tamper resistant devices conventionally used for securely storing keys and credentials and performing cryptographic operations. In a wide range of applications, smart cards are used to implement secure protocols with remote servers, in which secrets are shared between a remote server and the smart card. This is the case for example for mobile phone authentication, in which the smart card and server both share a set of secret keys for authenticating the user to the network.

In such end-to-end protocols, the server and smart cards are considered as trusted systems, whereas the host on which the card is connected is considered as un-trusted and acts merely as a gateway to the network. The host on which the smart card is inserted is generally referred as the terminal, and can be a personal computer, mobile phone, or point-of-sale terminal.

End-to-end protocols are used successfully for a wide range of applications such as end-user authentication or remote management of the content of the smart card. However, some classes of applications require that the smart card communicates with an application on the terminal without communicating with a server. Examples of such applications are a Voice-over-IP (VoIP) software client on the terminal that uses the smart card for user authentication, or digi-

¹ This work was supported in part by the European Commission (contract N° 021186-2 for the RE-TRUST project)
tal signature software on the terminal that uses the smart card for signing documents. In this case, it is important that the smart card can trust the terminal application, to prevent malware to perform VoIP calls or digitally sign documents without the user consent and knowledge.

Figure 1: End-to-end security with smart cards and servers – A secret is shared between the server and the smart card, and used to establish a secure channel. The terminal is un-trusted and is only a gateway for passing secure channel messages back and forth; the terminal does not share the secret and messages are opaque to the terminal.

The trusted platform module (TPM) is an attempt at solving the issue of trusting terminal applications. TPM typically manage keys and verify operating system components during startup to ensure that the terminal has not been tampered with. However, user credentials such as VoIP keys or public key infrastructure (PKI) keys are generally used by non-operating system components, and linked to the user identity rather than the terminal and operating system. TrustZone is an alternative technology, in which a security module is installed in the terminal to provide a security framework for applications. Both TrustZone and TPM require specific hardware and operating system support.

In this paper, we present how USB smart cards can be used to remotely entrust applications running on the terminal using white-box cryptography [WyseurB] and application monitoring, without requiring any specific security hardware, nor operating system modifications.

2 Remote Entrusting

In remote entrusting, an application is running on an un-trusted platform and sends requests to a trusted remote platform, typically a server [NagraJ,ScandariatoR]. The application execution is trusted with the collaboration of a local monitor embedded in the terminal, and of the trusted remote platform. The monitor logs local properties, such as tag sequences, memory thumbprints or execution time [CeccatoM]. The monitor logs are sent periodically to the remote trusted platform using a secure transmission protocol. The remote trusted platform analyzes the monitor logs, and track deviations from the expected logs. In case of tampering, the communication channel is closed and the service is stopped for the un-trusted application.
Figure 2: Remote entrusting architecture – A remote trusted server is providing a valued service, such as an online game, to a trusted client application executing on an un-trusted terminal. In parallel, a monitor executing on the client is monitoring the client application and sending some properties to a remote control service, or properties analyzer. Upon tampering detection in the properties, the properties analyzer instructs the trusted server to stop the valued service operation.

The monitor is a critical component in the architecture because like the client application, the monitor is exposed to attacks. The monitor is typically protected against tampering using code replacement techniques. Generally, the monitor is merged with the client application, which implies that the security provided by remote entrusting requires application providers to modify the client application.

Remote entrusting assumes that the client application is a remote client of an application service executing remotely on the network, otherwise no coercive control can done remotely by the trusted platform. In some variations of remote entrusting, such as in Figure 3, the control service can be delegated locally to a local control service executing on the terminal on trusted hardware, such as a smart card.
Figure 3: Remote entrusting using a local control service executing on a local trusted hardware, such as a smart card. The monitor is sending monitored properties both to the remote control service, and to a local control service, which can disrupt the client application execution upon tampering detection.

The remote entrusting principle has been extended to trust terminal applications from a smart card, which acts as the remote trust platform. Remote entrusting has been made possible by the recent evolutions of the smart cards.
Figure 4: Smart card entrusting of terminal applications executing on an un-trusted terminal. The smart card is providing some remote service to the terminal, such as cryptography (e.g. signing or ciphering) or data access (e.g. phonebook storage). The terminal trusted application embeds a monitor which reports execution properties to the properties analyzer executing on the smart card. Upon tampering detection from the properties, the properties analyzer executing on the smart card closes down the smart card services.

3 The USB Smart Card

Until recently, smart cards were devices communicating with the terminal thru a smart card reader using a half-duplex serial interface. The latest generation of smart cards communicates with the terminal using a high speed USB interface, which was developed to avoid the deployment of smart card readers, to improve the data transfer rate, and to provide new operating system interfaces in addition to the conventional smart card interfaces [AusselJD].

Because this new device is an extension of the classical smart card it is more appropriate to use another term like dongle or USB smart card to designate it.

Smart cards do not require any smart card readers on the PC terminal if they implement the Integrated Circuits Card Devices (ICCD) USB class standard, supported by most operating systems.

About the data transfer rates, the memory size of the smart cards increased of several orders of magnitude, from a few kilobytes to a few gigabytes, and high-speed USB allow fast data transfer that were not possible using the serial port communication.
Finally, the smart cards can now appear to the operating system as a mass-storage device (using the same interface as a memory stick, and appearing as a removable drive to the operating system), or as a network interface card using the USB Communication Device Class Ethernet Emulation Model (CDC-EEM). With CDC-EEM, smart cards can be accessed by terminal application using TCP/IP protocols. Typically, a USB smart card can appear to the operating system as a mass-storage device (read/write memory stick or read-only CDROM or both), a smart card reader for legacy cryptographic applications, and remote TCP/IP system.

In the remote entrusting model the trusted platform both provides an application service and controls the integrity of the application. By taking advantage from the evolution of the smart card this integrity role is delegated to the card. With this design a monitor sends application properties to the local control service located on the smartcard that is able to check for the integrity of the code.

![Figure 5: USB Dongle: an advanced smartcard – The dongle includes flash memory that will appear as removable drives to the operating system, and a conventional smart card. A controller is providing a mass-storage USB interface to the terminal, and managing communication between the terminal and the smart card and mass-storage](image)

4 Levels of Trust

A first basic level of trust is implemented with the read-only mass-storage partition of the dongle, which acts as a protected storage, and appears as a CDROM to the operating system. Since all applications located on the CDROM partition of the dongle are read-only, it is not possible for malware to tamper with these applications persistently. However, malware can still modify these applications on the fly, e.g. at load time or at run time.
Figure 6: USB smart card dongle — The flash memory on the dongle is partitioned as a CDROM partition and a private removable drive partition. The smart card is a conventional smart card communicating with the controller using an ISO7816 serial port interface. The dongle is viewed as three partitions by the operating system: a CDROM partition E:, a private partition F: which is only visible when a personal identification number (PIN) is correctly entered on the smart card, and finally a communication drive G:. Applications are communicating with the smart card using two files, an input file enchory.clp and an output file enchoro.clp. Trusted applications are stored on the CDROM partition and embed a set of keys shared with the smart card.

4.1 White Box Cryptography

In addition, white-box cryptography is used to establish a secure channel between the trusted terminal application and the smart card, and exchanging application runtime data measured by the monitor.

Each trusted application is stored in the dongle mass-storage partition, and loaded in the terminal memory for execution. Upon dongle insertion in the terminal, an opaque area is created in the application data segment in which a set of obfuscated keys are generated. These keys act as master keys to generate session keys for establishing a secure channel with the card. The master keys are computed when the dongle is connected to the host, that is before any application located on the dongle can be run. The application and the smart card must share a common secret that will be used for authentication during this new session keys generation.

4.2 Thumbprints

For this purpose a set of thumbprints is kept in a table by the card. A thumbprint is a section of the code segment that is hashed. Because an attacker may find which section of the code is taken as input for the hash, several thumbprints are processed during the dongle initialization. The smart card keeps all thumbprints in a table together with the corresponding code segment offset and size.
According to a heuristic, the session keys have to be changed after a certain period. Two equivalent keys set are computed both in the application and in the smart card. New session keys are derived from the master keys with a random value generated by the card and sent to the application.

Together with this random value, the card also sends both offset and size of a randomly selected thumbprint that will be used later for authentication.

The application first generates the new session keys. Then, it computes the thumbprint from the considered code section and uses the new session keys to cipher the thumbprint and prepare a message. This new message is send to the card that is able to check that the application has retrieved the right thumbprint and has computed the right keys. The communication between the application and the card is still secured by the current session keys. The new session keys replace the old keys upon successful thumbprint verification, first in the card and then in the actual application. This order is important to keep the coherence in case the user randomly and unexpectedly removes the card.

![Thumbprint mechanism.](image)

### 4.3 Block Size

Several options are available to select the number of blocks and the block size, leading to a compromise between higher security and performance. On one side, an option is to maximize the number of blocks. All the application code would be split in blocks of different size, and the corresponding hash values stored into the smart card. The size of the block is correlated with the security level, since the largest size is the whole code segment itself. But for performance reasons, it is not acceptable to hash the whole code segment at each session keys replacement. A compromise is to limit the number of blocks so that performance is not affected.
5 Limitations

There are still many security issues to solve with this design. First, the thumbprint mechanism is not tamper proof. An attacker can maintain a copy of the initial code segment to be executed only for session keys replacement and then switch to a modified one.

Still, the main weakness is that the application is sensible to attack on the master key values, which breaks everything in case of success. Current research apparently tells us that there is no ideal white-box cryptography algorithm to hide these keys with a high security level. That means that a determined attacker theoretically could always retrieve key values. Lifetime of the keys could be shorten to minimize this risk but still a more effective protection must be found to prevent this attack. One advantage of the current implementation is that master keys are not persistent, and generated randomly at each smart card insertion. An attacker retrieving a master key would only compromise the current session for the current token.

Another possible attack is that at a certain time the master keys are stored in clear in the memory during the key replacement process and in case of hole of security in the application’s code an attacker able to take control of the application could directly read the key values. Again there is no perfect protection except making applications more robust by raising the security limit high enough to discourage average attacker. At least when the key values are no more required, the memory buffer should be overridden with random data or other unrelated key values. Careful study of the generated code must be done to check that the optimizer does not remove the scrubbing code.

6 Conclusion

Remote entrusting with advanced smart cards allow to the execution of trusted applications on a terminal without requiring any specific hardware or operating system components. The level of trust is not as high as an end-to-end server to smart card connection, since it is based on white-box cryptography and is only as good as the key obfuscation algorithm. It is assumed that the user is not the attacker that is definitively a different context than the classical use of cards with a payment terminal. It enables however a better level of trust for mass-market applications such as voice-over-IP softphones or Internet authentication.

References


**Keywords**

Remote entrusting – USB – smart card