Use of Secure Device Identifiers in Virtualised Industrial Applications

MARCOS SIMÓ PICÓ
Abstract

Industrial Control Systems (ICS) running in a virtualised environment are becoming a common practice, however, there is not any standard or specification detailing authentication methods for industrial environments.

Considering the current standards and specifications designed to provide authentication, we present the design and implementation of several approaches that enable trusted computing in virtualised environments. Most of the approaches are based on a hardware-based root of trust, assuring the user’s software is always running on the same workstation.

After comparing the approaches, we test an efficient approach by using the SecDevID stored in the virtual TPM to establish TLS sessions. Given the TLS features, this approach provides both hardware and VM authentication as well as confidentiality. Finally, the performance of the tested approach is evaluated.
Abstrakt Begrepp

Industriella styrsystem (ICS) som körs i en virtualiserad miljö blir allt vanligare, men det finns hittills ingen standard eller specifikation för autentiseringsmetoder i industriella miljöer. Baserad på de gällande normer och specifikationer för att genomföra autentisering, vi presenterar design och implementation av flera metoder som möjliggör trusted computing i virtualiserade miljöer. De flesta av de metoder är baserade på en hårdvarubaserad ankare av förtroende, som garanterar att användarens mjukvara alltid körs på samma hårdvara. Vi jämför olika metoder, och testar en effektiv metod som avnäder SecDevID lagrad i en virtuell TPM för att etablera TLS förbändelser. Tillsammans med TLS ger lösningen autentisering för både hårdvara och VM, samt konfidentietät. Vi utvärderar prestandan av den sistnämnda metoden genom ett experiment.
Acknowledgements

I would like to acknowledge the following people, whose encouragement and support have been invaluable to me in the writing of this report: First and foremost, my parents whose support and motivation allowed me to enjoy the Erasmus experience in Stockholm; Rosa, my girl, who made the worst moments much easier. Thank you very much for your help and encouragement. To György and Valentino, for your patience and extremely useful and valuable help and feedbacks. This could not have been done without you two. To Emilio, Stelios, Sakis and the whole LCN in general. You made me enjoy the big amount of hours spent in the department (productive and non-productive at all). Thanks to the whole Bredäng group, specially to Gideon, Onno, Alex, Caroline, Sara, Álvaro, Lili, Cristophe, Julia, Christopher, Georgina, Hector, Lazare, Wael, Maria, Natascha, Verena, and Wieert. You made me remember Stockholm with so much affection. I wish I can see you all again. And my special thanks to the Hispanic crew in Bredäng, Kike, Miquel and Nico, you made me have great memories of this year. We spent together the best moments of the year.
# Contents

1 Introduction ................................................. 1  
1.1 Methodology .............................................. 1  
1.2 Ethical Considerations ................................. 2  
1.3 Report structure .......................................... 2  

2 Background .................................................. 3  
2.1 IEEE 802.1X - Port-based Network Access Control ......... 3  
2.2 IEEE 802.1AR - Secure Device Identity ................... 4  
2.3 Trusted Platform Module ................................. 5  
2.3.1 Trusted Platform Module 1.2 .......................... 5  
2.3.2 Trusted Platform Module 2.0 .......................... 7  
2.4 Virtual Machine Networking .............................. 7  
2.5 Xen Hypervisor ........................................... 8  
2.5.1 Virtualised Trusted Platform Module ................. 9  
2.6 ARM TrustZone® .......................................... 11  
2.7 Intel® SGX ................................................... 12  
2.8 Transport Layer Security ................................. 13  
2.9 GnuTLS Transport Layer Security Library ............... 17  
2.10 Enrollment over Secure Transport ....................... 17  
2.11 Integrity Measurement Architecture .................... 19  

3 Security Requirements ...................................... 20  

4 Approaches .................................................... 22  
4.1 VM Certification ........................................... 22  
4.2 Code Signing ................................................ 22  
4.3 Signature Chain ............................................ 23  
4.4 vTPM Signing ............................................... 24  
4.4.1 Storing Signing Keys in the pTPM .................... 24  
4.4.2 Storing Signing Keys in the vTPMngr ................ 29  
4.5 Intel® SGX .................................................... 29  
4.5.1 Key Generation Within an Enclave ................... 29  
4.5.2 Key Provided by a Service Provider ................. 30  
4.5.3 Observations ............................................ 31  

5 System Implementation ...................................... 34  
5.1 vTPM-TLS ................................................... 34  
5.1.1 Centralized TLS Key Distribution .................... 34  
5.1.2 vTPM-Stored Key ......................................... 35  
5.1.3 vTPM-Sealed Key ......................................... 40  
5.1.4 Integration with EST ................................. 42  

6 Performance Testing ........................................ 43  

7 Discussion ..................................................... 45  

8 Conclusions and Future Work .............................. 46  
8.1 Conclusions ............................................... 46  
8.2 Future Work ................................................ 46
1 Introduction

In an industrial control environment, there are a number of applications that can be moved into a virtual environment. Virtual machines provide many advantages to the ICSs, including less physical hardware, greater scalability, easier upgrades and forward compatibility, leading to a considerable cost reduction. Moreover, since virtualisation separates physical hardware to software, it eases maintaining legacy systems. As long as the Virtual Machine Monitor (VMM) is compatible with the physical hardware, hardware can be easily upgraded and the same software can be run without modifications. Furthermore, some virtualisation software provide physical server failure-proof systems [1]. In case a physical server failure occurs, this feature provides an automated process for restarting virtual machines that were running on that server, on another working station. These are some of the reasons why companies contemplate to migrate to a virtualised environment.

Regardless of the transformation virtualisation is bringing to the ICSs, challenges in ICSs remain the same. Most of the potential ICSs vulnerabilities found in a non-virtualised system will also be found in the system once it is migrated to a virtualised environment. Further, the fact that a system is virtualised is usually transparent to the system, as it operates the same way.

However, when trying to implement security in a virtualised environment, we find that there are neither standards nor specifications specially developed for virtualised scenarios. As stated in [2], “authentication is the base for several security mechanisms”, and specially in ICSs, a strong authentication mechanism is crucial. One important part of the authentication is identification, and it becomes more challenging in virtualised scenarios. Note that virtualisation allows several machines to be running on the same physical machine, making identification harder. Furthermore, by using this technology users are able to much more easily create and modify new machines, making identification even more important in virtualised scenarios. Thus, static secure identification methods become no longer feasible. For instance, identifying machines by LAN port is questionable, since multiple VMs may be running on a physical host.

While IEEE 802.1AR (Secure Device Identity) and the TPM (Trusted Platform Module) are two currently valid solutions for identifying physical devices, they are not valid for virtualised environments. We find a gap in the state of the art that fulfils virtual machine authentication. Considering the fact that in virtualised scenarios users can easily create and modify new machines, and aiming to fill this gap, we propose several ways to securely identify virtual machines. A hardware-based root of trust will assure us that the virtual machine has not been migrated to another physical machine, while a software entity will identify the specific virtual machine among all the existing virtual machines running on the same workstation.

1.1 Methodology

The first stage of this work was to study the current specifications and standards for authenticating devices in a non-virtualised scenario, including IEEE 802.1AR and the TPM. These concepts were extended in order to authenticate not only physical devices but also virtual devices (virtual machines). The current state of the art was reviewed and several theoretical solutions are detailed.
Finally, the current implementations of Xen hypervisor (and its virtual TPM feature) and GnuTLS (Transport Layer Security Library) are used together to implement a suitable solution, providing virtual machine and hardware workstation authentication. Following that, the performance of this solution was tested against a simple TLS authentication use-case in a non-virtualised scenario.

1.2 Ethical Considerations

The purpose of this work is to present authentication methods to be used in virtualised environments. In this work it is only detailed how to implement authentication methods, and it is meant to be used to avoid any kind of impersonation attack. Besides, the intended use of the tools used in this report is to provide authentication, and they are not meant to be used to perform any kind of attack.

However, note that in this report it is also detailed which of the considered approaches achieve the different security requirements (detailed in Section 3 and which ones do not. Except for R3, each security requirement can be easily associated to an attack vector. Thus, by using vulnerable approaches, an attacker could take advantage of this work to perform attacks exploiting the non-achieved requirements.

Please refer to section Section 7 for further information regarding security requirements achieved by the different approaches.

1.3 Report structure

The following sections of the report are organised as follows. Section 2 introduces a brief background on multiple key topics, standards, protocols and specifications, mentioned later in this report. Section 3 lists and details all the considered security requirements in this report. In section 4 the different considered approaches are presented. These approaches are: virtual machine certification, code signing, virtual TPM signing and Intel® SGX signing. Section 5 presents the details of our chosen approaches, joining the vTPM concept and TLS. Section 6 shows the results of comparing the performance of one tested approach versus the regular use of TLS. Section 7 summarizes and compares at a high level all the approaches, and finally, Section 8 concludes the report and details the future work that can be done in order to follow this report.
2 Background

This Section introduces a brief background on multiple key topics, standards, protocols and specifications, mentioned later in this report.

First Subsection details IEEE 802.1X (Port-based Network Access Control), which can be used in conjunction with IEEE 802.1AR (Secure Device Identity, detailed in the second Subsection) in order to authenticate network devices. The IEEE 802.1AR standard can be implemented with a TPM (Trusted Platform Module), a module currently included in many computers. TPM specification is briefly explained in Subsection 2.3.

Since the above-mentioned authentication standards and specifications are meant to be “exported” to a virtualised environment, Subsections 2.4 and 2.5 provide a concise background on this topic. Subsection 2.4 details current Virtual Machine Networking standards, while Subsection 2.5 discusses about the Xen Project, an Open Source hypervisor, focusing in the Xen’s virtual Trusted Platform Module implementation.

Thereupon, Subsections 2.6 and 2.7 detail hardware-based root of trust alternatives to the TPM: ARM TrustZone® and Intel® SGX. Both are trusted computing architectures based on the CPU.

A brief introduction of the Transport Layer Security (TLS) is detailed in Subsection 2.8. TLS can be a valid use-case of hardware-based keys (either TPM or CPU-based keys). In addition, Subsection 2.9 introduces the Gnu TLS library, an Open Source Implementation of TLS with TPM support. Later in this report it will be referred to this library as well as its API.

“Enrollment over Secure Transport” (EST) is briefly explained in Subsection 2.10. This standard details how digital certificates can be issued and managed over TLS, and this can be applied to certificates bound to TPM-stored keys.

Finally, Subsection 2.11 introduces the Linux “Integrity Measurement Architecture”, a TPM-based method to verify software integrity. Later in this report it will be used for “sealing” keys to a specific software status.

2.1 IEEE 802.1X - Port-based Network Access Control

Port-based network access control [3] specifies how a network administrator can restrict the use of IEEE 802 LAN service access points (ports) to secure communication between authenticated and authorized devices. This standard specifies a common architecture, functional elements, and protocols that support mutual authentication between the clients of ports attached to the same LAN.

The standard mandates the use of EAP (Extensible Authentication Protocol) to support authentication using a centrally administered Authentication Server, usually a RADIUS server. The standard also defines EAP encapsulation over LANs (EAPOL) to convey the necessary exchanges between peer Port Access Entities (PAE) attached to a LAN.

The authentication procedure involves three parties: a supplicant, an authenticator and an authentication server. A supplicant is an entity that is being authenticated by an authenticator. The Supplicant is usually connected to the Authenticator at one end of a point-to-point LAN segment. The term supplicant can also be referred to the software that communicates to the authenticator to gain authorization. The authenticator is an entity that requires authentication from the supplicant. Usually the authenticator is a network switch. The
authentication server is an entity that provides an authentication service to an authenticator. This service verifies from the credentials provided by the supplicant, the claim of identity made by the supplicant. Figure 1 illustrates a simple example of the location and the role of the three involved parties in a network.

IEEE 802.1X can be used together with IEEE 802.1AR since Port-based Network Access Control requires a secure identifier and credential in order to authenticate and establish trust in a device. The two standards are compatible.

2.2 IEEE 802.1AR - Secure Device Identity

The IEEE 802.1AR standard [4] specifies the secure Device Identifiers (DevIDs) and the management and binding of a device to its identifier. The DevId is a device identifier that is cryptographically bound to the device itself. It consists of the Secure Device Identifier Secret and the Secure Device Identifier Credential. DevIDs are designed to be used as secure device authentication credentials with standard authentication protocols such as EAP. An 802.1AR-compatible device incorporates a globally unique Initial Secure Device Identifier (IDevIDs), which can be generated internally in the DevID module, or by an external entity. An IDevID credential is, indeed, an X.509 credential. As an X.509 credential, it can be validated using the RFC 5280 defined mechanisms, and can be obtained after the DevID secret has been generated.

The device may support Locally Significant Device Identifiers (LDevIDs) by a network administrator or the device owner. As stated in [4], “each LDevID is bound to the device in a way that makes it infeasible for it to be forged or transferred to a device with a different IDevID without knowledge of the private key used to effect the cryptographic binding. LDevIDs can incorporate, and fully protect, additional information specified by the network administrator to
support local authorization conventions”. Moreover, LDevIDs should have the capability to be used as the unique identifier (by disabling the IDevID) to assure the privacy of the user of a DevID and the equipment in which it is installed. The DevID module shall include the IDevID and zero or more LDevIDs.

However, there are still some open issues in this standard [2]:

• The creation and use of LDevIDs requires the existence of a local certification authority.

• In case the IDevID is generated internally, the device has to communicate with the certification authority in order to sign the IDevID credentials. However, the standard does not specify any communication and signing process.

• In the standard it is specified that private keys must be stored confidentially and not available outside the module. Nevertheless, it is not specified how it has to be achieved.

• The IDevID is a standardized X.509 credential, usually with very long validity periods. Certificate revocation mechanisms should be defined in the standard.

• According to the standard specification, multiple logical devices may be contained within an aggregate device and, each of these logical devices will have its own unique DevID. However, the procedure to perform this is not specified.

DevIDs are used in conjunction with IEEE 802.1X to authenticate access to networks. The IEE802.1X standard is detailed in Section 2.1. At the same time, there is a close relationship between IEE802.AR and the Trusted Platform Module, detailed in section 2.3. Annex B of [4] provides a detailed explanation of how to implement a DevID with a TPM. The TPM specification fulfils most of the capabilities of a DevID module as defined by [4].

2.3 Trusted Platform Module

The TPM [5, 6] is a specification of a security hardware device defined by the Trusted Computing Group. The TPM is usually attached or soldered to the motherboard of the computer. The TPM, used as a hardware root of trust, provides us with secure storage and services even in case we do not trust the operating system an application is running on.

2.3.1 Trusted Platform Module 1.2

According to the TPM1.2 specification, the TPM consists mainly of a crypto-processor, a microcontroller specially designed to deal with cryptographic keys and operations, and some other components which are described below:

• Input and Output component: the I/O component is in charge of managing the information going to and coming from the communications bus. It routes the received information to the appropriate components and performs access control policies.
- Cryptographic Co-Processor: the cryptographic co-processor performs cryptographic operations within the TPM, such as asymmetric key generation, asymmetric encryption/decryption, hashing and random number generation. To do so, it is comprised of, at least, an RSA engine and symmetric encryption engine. It could also be supported AES as a symmetric encryption algorithm.

- Key Generation: the key generation component is responsible of generating RSA asymmetric keys pairs and symmetric keys.

- HMAC Engine: the HMAC engine component is in charge of calculate HMAC codes according to the RFC 2104. However, the RFC 2104 gives us the freedom of choosing the key length and the block size. Those parameters are defined in the TPM specification as follows: key length of 20 bytes and block size of 64 bytes.

- Random Number Generator: the Random Number Generator component is the source of randomness in the TPM. A good source of randomness is needed in several cryptographic algorithms and protocols, such as values of nonces definition and key generation. The random number generator consists of a finite state-machine and a one-way function (the SHA-1 engine can be used). It should provide 32 bytes of randomness on each call.

- SHA-1 Engine: SHA-1 is implemented in the TPM, as it is a trusted implementation of a hash algorithm. It should be implemented as defined by FIPS-180-1.

- Power Detection: this component manages the TPM power states. The TPM requires to be notified of every power state change.

- Opt-In: the Opt-In component provides the ability to allow the TPM to be turned on/off, enabled/disabled and activated/deactivated. It has several flags that indicate the state of the TPM. The setting of flags require authorization of the TPM owner.

- Execution Engine: this component executes the TPM commands received from the I/O port. It is the heart of the TPM. It ensures that operations are properly segregated and shield locations are well-protected.

- Non-Volatile Memory: non-volatile memory is used to store persistent state and identity associated with the TPM. However, it is also available for storage and use by authorized entities. Applications should avoid frequent writes of the same value, in order to avoid wearing out the device.

- Platform Configuration Registers (PCRs): A platform configuration register is a 160-bit register designed to store integrity measurements. The TPM specification states that a minimum of 16 PCRs must be present on a TPM chip. Since it is difficult to authenticate the source of measurement of integrity metrics, a new value cannot simply overwrite the previous value. PCR values are updated using the TPM_Extend command. Updating the PCRs results in a SHA-1 hash over the concatenation of the old value and the new generated value.
2.3.2 Trusted Platform Module 2.0

Trusted Platform Module 2.0 [7, 8] is a newer specification, which provides the same features as 1.2 plus some more [9]. After several years of using TPM1.2, the specification is updated in order to avoid the constraints on its use. Some of the extra features are described below:

- **Algorithm Agility:** unlike TPM1.2, TPM2.0 allows a lot of flexibility in what algorithms can be used. TPM1.2 can only use SHA-1 as hash algorithm, which is known to have security flaws [10]. According to the National Institute of Standards and Technology (NIST), “From January 1, 2011 through December 31, 2013, the use of SHA-1 is deprecated for digital signature generation. The user must accept risk when SHA-1 is used, particularly when approaching the December 31, 2013 upper limit. This is especially critical for digital signatures on data for which the signature is required to be valid beyond this date.” Although SHA-256 is the most used in early TPM2.0 designs, any hash algorithm can be used. Regarding the encryption algorithms, elliptic curve cryptography is supported in this new specification. In fact, TPM2.0 allows any kind of encryption algorithm, which algorithms are supported on a chip is a manufacturer’s decision. This means that in case an algorithm is found to be vulnerable, the specification will not need to change.

- **Non-Brittle PCRs:** With the TPM1.x keys and data can be sealed, meaning that are locked to certain PCRs values. This approach have some problems when updating the system, since PCR values are updated reflecting the modifications of the system. Therefore, all the secrets sealed to the PCRs that will be modified after the updating, have to be unsealed and sealed again after the process. However, the TPM2.0 specification allows you to seal data to PCR values approved by a signer, instead of to a determined PCR values. This way, sealed data can only be unsealed if the system is in an approved state by a particular authority.

- **Identifying Resources by Name:** in the TPM1.2 specification, resources are identified by their handles instead of by cryptographically bound names. It is detailed in [9] an attack performed exploiting this TPM1.2 feature: if two resources had the same authorization, and the low-level software could be tricked into changing the handle identifying the resource, it was possible to fool a user into authorizing a different action than they thought they were authorizing. In TPM2.0 this attack is not possible due to resources identification by name, cryptographically bound to them. Moreover, a name can be signed, providing integrity. In case the key is duplicated, this signature can be used to prove the TPM that created the key.

2.4 Virtual Machine Networking

Since server virtualisation is becoming an attractive solution among companies, the assumption that each network access port corresponds to a single physical

---

1 Although, in the TPM2.0 specification [11, 8] is referred to as handle.
device may no longer be valid. Virtualised servers running several virtual ma-
chines now transparently share the same physical server and I/O devices. We 
can handle VM networking in two different ways: we can implement a software 
switch as part of the hypervisor, switching the different VM’s packets as if the 
VMs were different entities. As an alternative, switching can be performed by 
an external switch.

The first solution results in what is known as Virtual Embedded Bridge 
(VEB) [12]. This approach (Ethernet bridge that resides within the hypervisor) 
might be fully standards-compliant with IEEE 802.1Q (VLAN). VEBs are often 
provided by hypervisor vendors, and managed through hypervisor management 
tools.

The second solution relies on an external hardware switch. All network traffic 
generated by any virtual machine is forwarded to an external switch. Within 
this approach, a further distinction can be made between tagless (reflective 
relay) and tagged (multichannel and port extension) options. These options 
are currently under development in the IEEE within the IEEE 802.1Qbg and 
802.1BR working groups. The use of an external switch has the advantage 
of consolidating the virtual and physical switching infrastructure into a single 
etentity and simplifying the management infrastructure.

All standards related to virtual networking are in the draft stage at the time 
of writing this report. However, networking vendors have developed proprietary 
solutions to meet today’s requirements.

2.5 Xen Hypervisor

There are two different kinds of hypervisors: type 1 hypervisors and type 2 
hypervisors. Type 1 hypervisors, also known as native or bare-metal hypervi-
sors, run directly on the system hardware. Type 2 hypervisors require a host 
operating system, and the host operating system provides I/O device support 
and memory management. Virtualisation started its first steps in type 2 hyper-
visors. Nonetheless, type 1 hypervisors are becoming more popular due to their 
superior performance. VMware vSphere as well as Xen Hypervisor are a type 1 
bare-metal hypervisor.

Xen is an open-source type-1 hypervisor firstly developed by the University 
of Cambridge Computer Laboratory in 2003 and later maintained and devel-
oped by a global large community (Xen project community). The Xen project 
policy is based on openness, transparency and meritocracy. Therefore, people 
participating in this project are earning responsibilities within the project as 
they more actively participate in the project.

Among the Xen hypervisor features, the following ones stand out:

- Support for multiple guest operating systems, including Windows, NetBSD, 
  FreeBSD and many Linux distributions.
- Support for multiple cloud platforms.
- Scalability (offers up to 4095 host CPUs with 16TB of RAM).
- Security: it has a dedicated security team, and offers multiple security 
  features such as VM introspection and vTPM.
- Is Open Source.
The Xen project (and the literature about it) uses specific nomenclature, different from other hypervisors. Thus, guest is also referred to as domain. Xen requires a management virtual machine, called dom0. Dom0 is the first VM started in the system, and Xen hypervisor is not usable without it. Dom0 has special capabilities like direct hardware access, system I/O handling and other VMs interaction. The “regular” guests are called domU. Although it was mentioned that Xen provides support for multiple guest operating systems, Xen requires dom0 to be a Linux paravirtualised guest. Most of today’s Linux distributions provide paravirtualisation-enabled kernels (including Debian, Ubuntu, openSUSE, SLES, XenServer, Gentoo Linux, Red Hat, Finnix, Oracle VM and Fedora). Furthermore, multiple distributions offer the Xen project software from their repositories, not needing to install the Xen project from source.

Xen hypervisor supports two different kinds of guests: Paravirtualised guests (PV) and Full or Hardware assisted Virtualised guests (HVM). Both type of guests can be used at the same time. Note that at the time of writing this report, only HVM is available for Windows guests.

Usually, Xen guests have access to one or more paravirtualised network interfaces. PV network interfaces are implemented with a couple of PV back-end and PV front-end drivers. The fron-end driver resides in the guest domain, while the back-end driver resides in Dom0. By opening additional channels of communication between the hypervisor and the domains operating systems (via PV front end and back end drivers), performance is improved since system’s resources do not need to be emulated. In most paravirtual-enabled kernels default drivers are available for PV network devices.

By default, Xen uses bridging within the Dom0 to allow domains to appear on the network as individual hosts. In this configuration a software bridge is created in Dom0 and backend virtual network interfaces are added to the bridge along with a physical Ethernet device. The bridge-utils package provides this utility. Once installed, one can configure the file /etc/network/interfaces look like it is detailed below to add the virtual eth0 interface of the guest to the software bridge, xenbr0:

```
auto lo
iface lo inet loopback

auto xenbr0
iface xenbr0 inet dhcp
bridge_ports eth0

auto eth0
iface eth0 inet manual
```

2.5.1 Virtualised Trusted Platform Module

The goal of the virtualised Trusted Platform Module (vTPM) is to, transparently for the virtual machines, enable trusted computing service to an unlimited number of virtual machines. With only one hardware TPM, every virtual machine is able to use it, just as if there were one hardware TPM for each virtual machine. Software written to interact with a physical TPM can run unmodified in a virtual environment with vTPMs, i.e., applications are unaware that they are actually accessing to an emulated device instead of an actual device. The
vTPM is implemented in a way that provides a strong association of the vTPM with the underlying hardware TPM.

In [13] the authors developed the software and protocols needed to implement virtualised trusted platform modules meeting, according to them, each of the mentioned features. The proposed architecture consists of a management virtual machine, which runs the vTPM manager. Every other virtual machine is able to have access to a vTPM instance. The vTPM instances are created, managed and deleted by the vTPM manager. The management virtual machine runs the server-side TPM driver meanwhile other virtual machines run the client-side TPM driver.

There must be a strong association between each vTPM instance and its corresponding virtual machine. In order to achieve it, it is attached a 4-byte vTPM instance identifier to each packet carrying a TPM command. A virtual machine cannot get access to a vTPM instance that is not associated with it. In order to maintain the association over the time, a virtual-machine-to-vTPM-instance table is created and maintained all over the time.

A virtual trusted platform module as a TPM able to spawn new vTPM child instances has been designed. It has been called vTPM root instance. This capability (the ability to spawn) should only be accessible to the owner of the root instance, i.e., the administrator of the management virtual machine. The TPM specification states that there has to be a storage root key (SRK) as the root key for its key hierarchy. Each generated key has its private key ciphered by its parent key. This way a chain is created to the SRK. In the vTPM instances this is done likewise, that is, an independent key hierarchy is created on each vTPM. Therefore, every vTPM instance is unlinked to the hardware TPM hierarchy. Key generation is, thus, much faster and, vTPM instance migration to other virtual machines is simplified.

According to the TPM specification, as stated in 2.3, every TPM has to have at least 16 Platform Configuration Registers (PCRs). A PCR is a 160-bit register, designed to store integrity measurements. PCRs are initialized at power up and can only be modified by extension (update the register) or reset. In the vTPM design, the lower PCR registers, which are defined as read-only registers, are used to show the values of the hardware TPM. The upper registers, which are read/write registers, reflect the specific values to each vTPM. Those measurements reflected by the upper PCRs include the hypervisor, boot process, BIOS and operating system, and they are specific to each VM. Using this attestation architecture it is achieved the vTPM-to-hardware-TPM linking. Thus, a challenger can check whether the measurements are the ones expected, meaning that the system has not been modified no upgraded without permission.

In order to implement the requirements specified above, the existing TPM 1.2 command set has been extended with the following additional commands:

- CreateInstance
- DeleteInstance
- SetupInstance
- GetInstanceKey/SetInstanceKey

¹Is meant in this context by attestation to confirm that some software or hardware is genuine or correct [13].
• GetInstanceData/SetInstanceData
• TransportInstance
• LockInstance/UnlockInstance
• ReportEnvironment

All the above-mentioned information regarding the vTPM implementation is in accordance to the IBM’s research [13]. In that research, the vTPM has been implemented and tested for Xen hypervisor. However, some features do not match the current Xen implementation. Further progress has been made by the Xen developers team regarding the vTPM feature since the IBM’s research was published, and at the time of writing this report, it still remains under development.

Due to this constant development, there are some differences between the architecture proposed in [13] and the vTPM implementation in Xen. Unlike it is explained in [13], vTPM instances are bound to the hardware TPM. Actually, vTPM instances’ data are stored within the vTPM manager encrypted with a pTPM key. This makes vTPM instance migration rather complicated, but in return, it makes the vTPM implementation more secure against VM migration attacks. Further, in the current vTPM implementation, lower PCRs are are not read-only, and are initialised with zeroes by default.

2.6 ARM TrustZone®

TrustZone® [14] is ARM’s contribution to Trusted Computing. Devices developed with TrustZone® fully support a Trusted Execution Environment (TEE), according to the Trusted Base System Architecture. Basically, TrustZone® enables two virtual processors on every CPU core: the Normal World and the Secure World. According to the design principles, the Secure World manages the security subsystem, while everything else is managed by the Normal World. The task of switching from one to another virtual processor or world is performed by the Monitor Mode. It is, indeed, the interface between the two worlds. The physical processor can enter from Normal to Monitor Mode only in case a few situations are met. Moreover, once in Monitor Mode, interruptions are disabled for security reasons.

The Normal World components cannot access to the logic hardware present in the Secure World. Even the keyboard typing, display and touch-screen (and in general, every I/O peripheral) eavesdropping is prevented if software is running in the Secure World. Each virtual processor has access to its cache memories, which have additional tags to differentiate content cached by the normal and secure world. In addition, each virtual processor is provided with its own memory management unit in order to distinguish to which world belongs every page. Nevertheless, code running in the Secure World can directly access the Normal World components.

According to [14], “Many attackers attempt to break the software while the device is powered down, performing an attack that, for example, replaces the Secure world software image in flash with one that has been tampered with. If a system boots an image from flash, without first checking that is it authentic, the system is vulnerable.” Therefore, before booting the device, we must ensure its
legitimacy. A secure boot schema is facilitated in the TrustZone® specification, using cryptographic signatures. A chain of trust rooted on the SoC (system on chip) is implemented. Every stage is integrity-checked before executed. The boot sequence has seven stages: device power on, ROM SoC bootloader, flash device bootloader, secure world OS boot, normal world bootloader, normal world OS boot and system running. It is recommended to store the public key to verify the signature in the on-SoC ROM, since it is the only component that cannot be easily modified or replaced. However, this implies that all devices use the same public key. On-SoC One-Time-Programmable hardware, such as poly-silicon fuses is highly recommended to store unique values in each SoC.

Malware running in the standard OS can neither interfere code running in the Secure World, nor get access to the Secure World’s stored data. Even though isolation from one World to the other is well-defined, it is not from different applications running within the same world. Therefore, an application running in the Secure World (secure virtual processor) has access to the Secure World’s hardware. Exploited vulnerabilities of software running in the Secure World could lead to compromise secrets. Readers are referred to [14] for more information about TrustZone®.

We may think of several approaches in order to perform a secure signature. On one hand, the application generating messages to be signed would be running on the Normal World. Then, the application would ask code running in the Secure World to sign the messages with a private key stored within the Secure World. Finally, signature would be sent back to the application. The main issue of this approach would be application authentication. We cannot assure the application has not been compromised nor it is a legitimate application, i.e., one would not notice if it is a third party’s application, impersonating the legitimate one. Thus, we cannot consider this approach acceptable.

Another approach would be to consider the whole application running in the Secure World. Therefore, whenever the application needs to sign a message, it has direct access to the private signing key, stored in the Secure World. Unfortunately, as explained above, other applications or code running also in the Secure World could theoretically get access to the private key. The security of this approach and the difficulty of stealing the private key relies, of course, on the actual implementation.

Another limitation of ARM TrustedZone® is the hardware constraints we are compelled to. As it is explained in [14], the ARM architecture includes support for multiprocessor designs with between one and four processors in a cluster. Nonetheless, four processors means only one chip with four cores. Therefore, if a more powerful system is needed, any TrustedZone®-based solution does not fulfil our requirements.

2.7 Intel® SGX

Intel® Software Guard Extensions (SGX) [15] are a set of instructions and mechanisms for memory access which will be added to future Intel® processors. SGX allows applications to instantiate a protected area within the application’s address space, named in the literature enclave. The enclave provides confidentiality and integrity even in the presence of privileged malware. Enclave data is protected by the CPU access control. Furthermore, these data is encrypted and integrity checked when it is moved from the Enclave Page Cache (EPC) to
memory. It can be encrypted either using an enclave-specific or platform-specific key. Thus, larger amounts of data can be securely stored, and optionally, shared with other enclaves. Access to the enclave memory area from any software not resident in the enclave is prevented. SGX was designed to enable trustworthy applications to protect specific secrets or sensitive data from privileged software, in our case, the hypervisor and/or the VM’s OS. This data as well as some portions of the code can be securely stored in the enclave.

At manufacturing time, every Intel® SGX-provided processor is provided with a cryptographic key. This key is the basis of every other key generated by the CPU, the root of the key hierarchy. Therefore, an enclave requesting a key using the EGETKEY instruction will get a key derivate from the root key.

In addition, SGX has the capability of generating identities for enclaves. While the enclave is built, two identity values (measurements) are generated and recorded before enclave execution starts. Those values are MRENCLAVE and MRSIGNER. MRENCLAVE is the value identifying the enclave, and is the result of a SHA-256 hash operation. The hash is performed over an internal log register, which contains information of the code, data, stack, heap and security properties of the pages. Any change on the software would result in a different value of MRENCLAVE. At the same time, every enclave has a second identity, also known as sealing identity. It includes a sealing authority, a product ID and a version number. Usually the sealing authority is the enclave builder itself, and it signs the enclave prior to distribution. In case the MRENCLAVE value matches an expected value, the value specified in the enclave certificate (SIGSTRUCT), a hash of the public key of the sealing authority is stored in MRSIGNER register. This value can be used to seal data. Moreover, enclaves sharing the same sealing authority can share and migrate their sealed data.

Moreover, several enclaves can be instantiated at the same time, so different sensitive information can be stored into different enclaves. Note that the code running inside an enclave can read data stored in the enclave. Even though a particular application is benign, it may contain vulnerabilities that might be exploited.

Furthermore, Intel® has introduced an extension to the Direct Anonymous Attestation scheme used by the TPM [7] in order to avoid privacy concerns. This mechanism is called Intel® Direct Anonymous Attestation scheme (EPID) [16], and is used by the Quoting Enclave to sign enclave attestations, detailed in section 4.5.2.

Unlike TPM and ARM TrustedZone®, SGX was designed with virtualisation in mind, making it straightforward to apply this solution to a virtualised scenario.

2.8 Transport Layer Security

Transport Layer Security (TLS) [17] is a protocol defined by the IETF and designed to provide confidentiality and data integrity for process-to-process communications. The current version of this standard, at the time of writing this report, is TLS 1.2. TLS 1.3 is still a draft. TLS is the successor of SSL, the Secure Sockets Layer protocol, designed by Netscape. TLS is composed of two layers: the TLS Record Protocol and the TLS Handshake Protocol. On one hand, the TLS Record Protocol is running at the lowest level, over a reliable transport-layer protocol, usually TCP (described in detail in [18]). On the other
hand, over the TLS Record Protocol is the TLS Handshake protocol. The location of the TLS protocols in the TCP/IP model is shown in Fig. 2.

The TLS Record Protocol provides confidentiality and integrity-checked communications. In order to provide confidentiality, symmetric cryptography is used for data encryption, and data integrity is provided by the use of message authentication codes (MAC). For each connection, the symmetric keys are negotiated by another protocol (usually TLS Handshake Protocol) and uniquely generated. Integrity is also provided by the use of keyed Message Authentication Codes (MAC). Optionally, the TLS Record Protocol can fragment the data to be transmitted into convenient blocks, compress the data, apply a MAC and encrypt. Once the pertinent operations have been performed, the result is transmitted.

The TLS Record Protocol is used to encapsulate higher-level protocols. Among those protocols are the TLS Handshake Protocol, the Alert Protocol, the Change Cipher Specifications Protocol, and the Application Data Protocol. The operating environment of the current TLS protocol is defined as the TLS Record Protocol connection state. The security parameters of a TLS connection state are defined by the following values:

- **Connection End**: It defines whether the role of the entity is “client” or “server” in the connection.
- **PRF algorithm**: It defines an algorithm to generate keys from the master secret.
- **Bulk encryption algorithm**: This algorithm defines the symmetric encryption algorithm, including the key size and the type of encryption algorithm. That is, block cipher, stream cipher, or Authenticated Encryption with Associated Data (AEAD). If appropriate, it also specifies the size of the block and the size of the initialisation vectors (IV) or nonces. The algo-
algorithm can be chosen among RC4, 3DES, AES or none in case one does not want confidentiality in our communications.

- **MAC algorithm:** Defines the algorithm used for message authentication as well as the size of the MAC. The MAC algorithm can be chosen among HMAC MD5, HMAC SHA-1, HMAC SHA-256, HMAC SHA-384, HMAC SHA512 or none.

- **Compression algorithm:** It specifies all the needed information to compress data.

- **Master secret:** A 48-byte shared secret between the two end entities.

- **Client random:** A 32-byte random value provided by the client.

- **Server random:** A 32-byte random value provided by the server.

The record layer protocol uses the above security parameters of the connection state to generate the MAC keys, the encryption keys and the IVs when they are needed. The current states are updated for each record processed.

The record layer receives data blocks of arbitrary size from higher layers. These data are fragmented into $2^{14}$ bytes (or less) long blocks. Several higher level messages may be merged into one single message and vice versa. The compression algorithm, defined in the connection state, converts TLSPlaintext structure into TLSCompressed structure. More detailed information about compression in TLS can be found in [19]. A TLSCompressed structure is translated into a TLSCiphertext by the MAC and encryption functions. Moreover, the MAC contains a sequence number. Thus, any missing, repeated or extra message is detected.

The TLS Handshaking Protocol consists of three TLS subprotocols (Handshake Protocol, Change Cipher Spec Protocol and Alert Protocol). They allow client and server mutual authentication and symmetric encryption negotiation. During the negotiation, the encryption algorithms and encryption key features are decided. This negotiation must be done before the higher-level protocol transmits any message (any sent message before the negotiation would not be encrypted). Authentication is achieved taking advantage of public key cryptography. The negotiated secret is transmitted preserving confidentiality, since it is encrypted with the public key of the receiver. Any man-in-the-middle/eavesdropper can not obtain the secret. An attacker can not even modify the negotiation communication, given that it is integrity-checked. The Handshaking Protocol, responsible of the negotiation session and the security parameters establishment, consist of the following items:

- **Session Identifier:** it is an arbitrary value, chosen by the server to identify an active session.

- **Peer Certificate:** it is an X.509 certificate of the peer. It might be null.

- **Compression Method:** it is the algorithm used to compress data.

- **Cipher Specifications:** it specifies the pseudorandom function, the symmetric encryption algorithm and the MAC algorithm.
- Master Secret: it is a 48-byte secret shared between the client and the server.

- Is Resumable: it is a flag indicating if the session can be used to initiate new connections or not.

These items detailed above are used to create the security parameters used by the Record Layer. The resumption feature allows to instantiate several connections using the same session.

Alert messages, sent by the alert protocol, tell about the severity of the messages, whether it is a warning or a fatal error, and a description of the alert. Alert messages with a fatal result lead to an immediate termination of the connection. Nevertheless, other connections corresponding to the same session may not be interrupted. Alert messages are compressed and encrypted as specified by the connection state. Whenever an entity detects an error, it sends an error message to the other party. Upon reception or transmission of a fatal alert, both parties immediately close the connections and forget the session identifiers, secrets and keys related with the failed connection. At the same time, if a warning alert is sent or received, the connection may continue. However, if the receiving party decides not to proceed with the connection, given the warning alert, it should send a fatal error and terminate the connection. When a party detects a malfunction, it decides whether the alert is warning-level or fatal-level alert.

The Change Cipher Spec protocol is responsible for signal transitions in ciphering methods. The protocol consists of one single message, encrypted and compressed according to the current connection state. The message can be sent by both parties, to notify the receiving party that the next sent records will be secured by the newly negotiated specifications.

Lastly, the TLS Handshake Protocol is the responsible of generating the parameters of the session state. Basically, the Handshake Protocol involves:

- Hello messages exchange to agree on the algorithms. Protocol Version, Session ID, Cipher Suite, and Compression Method are established during this step.

- Server and client random values are exchanged.

- Cryptographic parameters exchange to agree on a premaster secret. This step uses up to four messages: the server Certificate, the ServerKeyExchange, the client Certificate, and the ClientKeyExchange.

- Digital certificates exchange to allow client and server authentication.

- Master secret generation from the premaster secret and the exchanged random values.

- Security parameters provisioning to the Record Layer.

- Application data exchange.

TLS offers the chance to resume older sessions established between two entities. When a connection is established by resuming a session, new ClientHello.random and ServerHello.random values are hashed with the older session’s master secret. Given that the older session’s master secret has not been
compromised, and that secure MAC keys and secure hash algorithms are used, the new session should be secure and independent from previous connections. That is, even if an attacker knew the previous session encryption keys or MAC secrets, the master secret cannot be compromised.

Note that sessions can only be resumed if and only if both client and server have the "Is resumable" flag set to True. If either entity suspects the session might have been compromised, a full handshake should be performed. It is suggested in [17] that an upper limit of 24 hours for session ID lifetimes is set. This is because in case an attacker obtains a master secret, it might impersonate the compromised entity until the session ID is retired.

TLS is a possible use case of SecDevID and TPM. TLS is a mature and widely-used technology and an attractive solution for many purposes. The session resuming feature can be specially interesting when using a TPM for storing private keys used later for establishing TLS sessions. Note that performing operations with the TPM are much more time-consuming than performing them directly on the CPU. Resuming sessions can notably reduce the number of TPM access operations, since the master secret of the previous session is re-used in order to generate a new one. This way it is improved the overall performance.

2.9 GnuTLS Transport Layer Security Library

The GnuTLS [20] is an Open Source implementation of SSL, TLS and DTLS (Datagram TLS). Basically, DTLS enables TLS to work over a non-reliable transport layer protocol, usually UDP. Therefore, DTLS also implements packet retransmission and sequence number assignment. SSL/TLS are explained in Section 2.8. Readers are referred to [21] for further information about DTLS. GnuTLS is, indeed, a library providing a simple C language application programming interface (API). The most important features of GnuTLS, described in detail in [20], are:

- Support for TLS 1.2, TLS 1.1, TLS 1.0 and SSL 3.0 protocols.
- Support for Datagram TLS 1.0 and 1.2.
- Support for handling and verification of X.509 and OpenPGP certificates.
- Support for the Online Certificate Status Protocol (OCSP).
- Support for password authentication using TLS-SRP.
- Support for keyed authentication using TLS-PSK.
- Support for TPM, PKCS #11 tokens and smart-cards.

2.10 Enrollment over Secure Transport

Enrollment over Secure Transport is an IETF standard, defined in [22]. It details certificate enrollment using Certificate Management over CMS, also known as CMC [23], over a secure transport layer. Enrollment over Secure Transport (EST) is extensible and may add additional features to CMC. Two extensions are defined in [22]: Certificate Signing Request attributes requesting and server-generated keys requesting. Each EST service or operation is accessed by a
different path-suffix, following the path-prefix of "/.well-known/" (as defined in [24]), and the registered name “est”. Therefore, a valid HTTP request-line for requesting CA certificates would be:

```plaintext
GET / . well-known/est/cacerts HTTP/1.1
```

EST runs on top of HTTP, which will usually be running on top of TLS. The general EST client/server interaction is performed as follows:

- The EST client initiates a TLS session with an EST server.
- The EST client requests a service from the server.
- The client and server are authenticated.
- On one hand, the client verifies that the server is authorized to serve it.
- On the other hand, the server verifies that the client is authorized to use the server and the service requested.
- The server acts according to the client request.

By submitting an enrolment request to an authenticated EST server, the client can get a certificate for itself. Previously, the client will have to be authenticated too by using either certificate or certificate-less method. This authentication can be either TLS or HTTP-Based. TLS is the recommended method for authorizing client enrollment requests, and it may use existing certificates. These certificates may have been issued under a distinct PKI, such as a certificate proving ownership of a TPM key or an IEEE 802.1AR IDevID credential.

In [23] the POP (Proof-of-Possession) concept is defined as “a value that can be used to prove that the private key corresponding to the public key is in the possession of and can be used by an end-entity”. The EST signed enrollment request provides a signature-based POP. Furthermore, in [22] is detailed how to link and end-entity ID and POP information. This is achieved by including specific information about the current TLS session within the signed certification request. The EST server may or may not request linking of identity and POP. According to [22], regardless of the EST server requests, clients should always link identity and POP by embedding TLS-unique information in the certification requests. The TLS-unique value is placed in the certification request “challenge-password” field, and is base64 encoded.

Before processing any request, an EST server checks the client authorization. The same way, the client determines if the EST server is authorized. Thereupon, the EST client will usually want to request a copy of the current CA certificates. Even if the client has not been configured with an implicit trust anchor database, that allows a bootstrap installation of the explicit trust anchor database, the CA certificates can be securely retrieved. In this case, the initial TLS server authentication will fail, since the client does not have the server’s CA certificate. Provisionally, the client can finish the unauthenticated handshake and extract the HTTP content data from the response. Then, a human user can authorize the CA certificate using out-of-band data such as the CA certificate “fingerprint”. This response will establish an explicit trust anchor database for the following TLS authentication of the server.
It is recommended for EST clients to request the EST CA trust anchor database information of the CA before stored information expires. This way, the EST client CA trust anchor database can be up to date. Anyway, according to the standard, EST servers should not require client authentication or authorization to reply to this kind of requests. Thus, even if the client’s stored data has expired, it will be able to retrieve the new database.

In response to this request, the server replies with an HTTP 200 code, if successful. The response is a certs-only CMC response, containing the root CA certificates and any additional certificates the client would need to build a chain from an EST CA-issued certificate to the current EST CA trust anchor.

Once the client is provided with an EST CA trust anchor database, mutual authentication can be performed between EST client and server. A simple enroll and re-enroll request, if successful, will be replied just like CA certificate request explained above, with an HTTP 200 code, a certs-only CMC response, containing only the certificate that was issued.

2.11 Integrity Measurement Architecture

The Integrity Measurement Architecture (IMA) is a TPM-based security improvement of the Linux Kernel. It was developed by IBM research and included in the Linux kernel main source tree since version 2.6.30. Its main purpose is to generate a verifiable value, representing software stack running on a Linux system. This value can be used for integrity-checking the system both remotely and locally.

The Linux kernel measures each executable, library, or kernel module loaded into before they can affect the system. The measurement consists of a SHA-1 hash of the files. Every measurement since system start up is stored in a kernel-held measurement list. This way, the SHA-1 hashes reflect the status of the system, letting know whether the system has been changed or not.
3 Security Requirements

Our goal is to provide security in virtualised scenarios. The approaches and solutions presented in this report will try to meet the following requirements:

R1: Strong Authentication. In an industrial control environment, either virtualised or not, one of the most important security mechanisms nowadays is a strong and reliable authentication[25]. Both the hardware and the software need to be authenticated. The objective is to assure that the messages received by another node are actually coming from that node, i.e., the node has not been impersonated. In other words, it is important to make sure that the message is actually coming from the station it claims it is coming from.

R2: Integrity. It is not sufficient making sure that the sender of the message is the one it claims it is, but we also need to make sure we can trust its content, i.e., the message has not been modified on its way to the receiver.

R3: Low-Latency. Not only cyber-security shall be considered in an industrial control scenario, but also performance and safety. For instance, in substation automation and inter-substation protection communication, delays must be limited to a few milliseconds [26]. Time constraints are a ubiquitous fact in this environment, and cyphering and deciphering delays are a crucial factor that must be considered. Therefore, not only the cyber-security level provided by different has to be considered. It is actually a trade-off between security and performance.

R4: System (software and hardware) trustworthiness. One must also verify that the applications are legitimate and assure that the applications have not been compromised. Moreover, one should check that neither the applications nor the hypervisor has been modified/updated without explicit permission. Note that in this scenario, authenticating and verifying the applications is not enough. Even verifying the authenticity and integrity of the whole stack of software running on the hardware machine, it could be duplicated or moved to another hardware. This duplicate copy managed by an attacker, would successfully pass the integrity verifications. That is the reason why not only software but hardware system must also be authenticated.

R5: Resilience to Replay Attack. Achieving the above-mentioned requirements, an attacker would be able to keep a copy of the message and retransmit it later. The message would pass the integrity checks (it would have not been modified) and as well as the authentication checks (the message would have been generated by the legitimate sender), however, it is not the sender who actually sent the message. Our approaches should consider this and avoid this kind of attacks. Note that this requirement can be implemented at the application level, regardless of the underlying approach.

Specially in a virtualised scenario, authentication mechanisms (R1) can be applied at several layers. Following a bottom-up approach, on the lower layer there is the system hardware. Then there is the hypervisor, which in our scenario
would be a type 1 hypervisor. Next layer there is the virtual machine layer and, finally, software or applications installed and running on the virtual machines.
4 Approaches

Digital signature is the cryptographic mechanism capable of providing authentication, non-repudiation and integrity. Therefore, by properly using digital signatures, one is able to meet requirements R1, R2 and R4 (detailed in Section 3). Missing requirements (R3 and R5) can be achieved at a higher layer.

In this section we describe several approaches meant to achieve all the requirements explicated in the previous section. More simple and straightforward approaches are explained first and more complex approaches are detailed later.

4.1 VM Certification

In order to verify and identify a VM, a certificate bound to the VM’s image can be generated. This certificate would uniquely identify the VM. Considering it is taken into account the whole VM image upon certificate generation (i.e., the certificate consists of VM ID information and a signed hash of the VM image), apart from identifying the VM one can also make sure that it has not been modified and does not contain any malicious software (since no software can be installed without modifying the VM image).

Every time a change occurs in the VM’s code (even a legitimate change), its hash would have to be recalculated and the certificate would have to be generated again considering the new VM value. This solution is computationally expensive, even infeasible for many scenarios. Further, the hash value and the certificate recalculation should be done frequently. The more frequent this check is performed, the sooner we would get notified of an illegitimate modification in the VM. Considering this check unfeasible, the VM’s certificate would be checked only once, while uploading the VM image. Therefore, we need to certificate the software installed in the virtual machine separately, making sure that is legitimate software. This can be done the way it is done in non-virtualized environments by using the Code Signing approach, detailed in subsection, 4.2.

Apart from verifying the trustworthiness of a VM by using its certificate, one may want to install a SecDevID (or in general, a private key and its corresponding certificate) in the VM image, being the different VMs running on the same hypervisor identified as different devices. Once identified, network access is controlled by 802.1X.

Note that by using this approach, the hypervisor needs to be trusted, since we cannot hide the SecDevID to it. A corrupted hypervisor could impersonate other the VMs it has running on top of it. Furthermore, the process of uploading/installing the VM needs to be trusted too, since the VM integrity-check must be performed.

By using this approach, we would not notice whether a VM is a legitimate one or an illegitimate copy (running either in the same system with the same hypervisor instance, or in another physical machine). Note that authentication is not applied at the hardware level.

4.2 Code Signing

Code signing is nowadays a very common practice. From the user point of view, code signature procedure is usually as follows:

- User downloads the software he/she expected to install.
• User download the code signature.
• User checks the signature file against the downloaded code. A successful signature check proves that the downloaded software has not been tampered with.

The main idea of this approach can be applied to our virtualised scenario. We consider that each application is specifically compiled for each VM, and the VM's private key and certificate have been hard-coded in the application. We also consider that the VM would have installed another key at the OS level.

Rather than checking the author of the code and its trustworthiness, we also want to make sure it is only installed in the VM the code is meant to be installed in. Thus, the code signing would be performed the other way around: the code would be compiled and signed with the public key of the VM (the one stored in the OS), and the VM would check the signature upon software installation, detecting whether the code is meant to be installed in that specific VM or not. The key pair in the application is intended to be only used to sign messages once the application is running, while the key pair in the VM is meant to be used to check the code prior to installation.

Considering that in the OS it is implemented a procedure to only allow installation and upgrades after the signature checking, this approach would assure that the application specifically compiled for one VM, can only be installed in that VM and nowhere else. Therefore, messages signed by the application would be actually coming from the VM it claims it is.

Note that in this approach, the process of installing certificates and private keys in the VM, the process of hard-coding keys and certificates as well as compiling the code needs to be trusted. The hypervisor (having access to everything in every VM) needs to be trusted too.

As happens with the VM certification approach, the whole VM could be illegitimately migrated or copied to another system and this migration could not be detected. In order to make this illegitimate migration attack noticeable, there is a need to somehow bound the VM to the hardware.

4.3 Signature Chain

Any signing approach providing integrity and authentication used in non-virtualised scenarios can be used in virtualised scenarios. Note that the fact that a system is virtualised can be transparent for the system itself. By checking the VM's signature one is able to identify the VM, and by verifying the VM's signature, the same advantages and drawbacks we had in the previous two approaches are arised, i.e., VM is authenticated, but it could be illegitimately migrated not affecting the behaviour of the approach.

In order to bind a VM to the hardware system, one can attach another signature to the message-and-VM's-signature block. The Trusted Platform Module seems to be the ideal device for attaching a hardware-rooted signature to this block. Therefore, at the end of this process one would have the message itself with the VM's signature attached, and then, this whole block would be signed with the TPM and finally sent. This structure is shown in Figure 3.

A certificate needs to be issued for the TPM signing key, being used to verify every VM running on the same physical machine. The same occurs for every VM, another certificate needs to be issued for each of them. Considering the
receiver has both certificates, first it would verify the message and VM signature block by checking the TPM signature. Once the TPM signature has been successfully verified, the message needs to be verified checking the VM signature. If successful, VM and physical machine would have been authenticated. Moreover, both signatures prove message integrity too.

This approach can be extended as one can add as many signatures to the message as one considers convenient, according to the application. Signatures can be added at the application level, VM’s Operating System, hypervisor and physical machine. However, for most of the cases, one software and one hardware signature can be considered enough. Note that one of the requirements described in Section 3, R3, is about time delays constraints. The more signatures we add the more delay we are adding too. The same occurs for the receiver, who has to check more signatures.

The main drawback of this approach is that it differs from the regular signature use. We wish that as much software as possible that was originally written to be used in non-virtualised scenarios can run unmodified in our virtualised system. Therefore, even though this solution is valid, both the sender and the receiver need to be aware that they are running virtualised, and therefore at least two signatures need to be performed.

Further, no more than one VM can be accessing the TPM at a time, and the hypervisor would need to be implemented considering this fact. Therefore, the only way to apply this solution is by using the vTPM. However, more efficient solutions using the vTPM are explained below.

4.4 vTPM Signing

4.4.1 Storing Signing Keys in the pTPM

Notation  We will use the notation according to [27], in our particular case, we are using the following notation:

- AES key for encrypting the vTPM data, shared secret between the vTPMmgm and pTPM: AES-$K_{vTPMmgm,pTPM}$.
- Public part of the TPM’s storage key: $Stg-K_{TPM}$.
- The private part of an asymmetric key is represented as the inverse of the public part: $Stg-K_{TPM}^{-1}$.
- Asymmetric VM’s signing key: $Sig-K_{VM}$.
- Message that the application wishes to sign: M.
- Digest or hash of the message M: $H(M)$.
- Handler of a key K: *K.
- Encryption of the message M using the key K: K(M).
Overview We consider a system that consists of a virtualised system, with several virtual machines running on it. The hardware of the system has a Trusted Platform Module (TPM)\cite{9} attached to the motherboard. The virtualisation software running on the system is XEN, an open source hypervisor. Since virtual machines need to get access to the TPM, the TPM has to be virtualised. Many VMs accessing concurrently at the same time to the same TPM is unfeasible. In order to allow concurrent access to TPM services by different virtual machines, XEN implements TPM virtualisation. The virtualisation of the TPM in XEN implies a stub-domain\footnote{See \url{https://blog.xenproject.org/2012/12/12/linux-stub-domain} for more information.} running the vTPM manager (vTPM-mgr) and one more stub-domain for each virtual TPM (vTPM). A vTPM is needed for each of the virtual machines (VM) willing to get access to the TPM. A schema of the system can be seen in Fig. 4.

The vTPMmgr is responsible for the management of the vTPMs as well as for the communications with the physical TPM (pTPM). Some TPM commands\cite{8}, such as \texttt{TPM2_PCR_READ()} will just pass thorough the vTPMmgr without getting any modification at all. The vTPMmgr stores all the vTPMs’ information. Among that information, there is the vTPM data, an RSA signature key handler, encrypted and sealed to the pTPM and to the state of the system (using the PCRs), and a hash of the TPM data. The vTPM data, the signature key handler and the digest constitute an \texttt{RSA\_BLOCK}. This \texttt{RSA\_BLOCK} is encrypted by the pTPM using a storage key. A schema of the vTPM data structure can be seen in Fig. 5 \footnote{As it is described in the XEN vTPM description, vTPMs’ secrets are stored on the vTPM data. However, we prefer to store on the vTPM data only the key handler, storing the actual key in the pTPM for security reasons.}.

We want to achieve VM and physical system authentication. The procedure to achieve our goals would be the one explained below. A schema of the messages exchange can be seen in Fig. 6, where messages are numbered from 1 to 10:

1. The application willing to sign a message M would perform a hash over M, H(M). Then it would invoke the \texttt{TPM2\_Sign()} command, sending as
Figure 5: Schema of the vTPM data stored in the vTPMmgr.

input parameter

\[ \text{msg1: } H(M) \].

The command is sent through the TPM frontend driver, which provides the standard TPM interface. A listing of the command’s parameters is provided in Listing 1 in the Appendix. The virtualised environment is, thus, transparent for the application, which believes that it has direct access to a TPM. The message is forwarded from the TPM frontend driver to the corresponding vTPM stub-domain. A UUID is generated in order to identify every vTPM and, on the other hand, a domain ID number is set to each VM at the time of its creation. This domain ID is an integer number, unique for every VM. It starts from zero and is incremented whenever a new VM is instantiated. Therefore, if we had the dom0 and 20 more VMs, their domain ID would be 0 for dom0 and 1-20 for the rest of VMs. The VM-vTPM association is stored in the file /var/vtpm/vtpm.db.

2. The vTPM stub-domain then forwards the same message to the vTPMmgr. The UUID of the vTPM instance is sent in the tpmfront_dev variable.

\[ \text{msg2: } H(M), \text{UUID}. \]

3. The vTPMmgr sends the standard TPM2_RSA_Decrypt() command to the pTPM through the frontend driver. The parameters of this command are, among others, *Stg-KTPM (The handler of the needed key to decrypt the block) and the RSA_BLOCK linked to the vTPM wanting to sign. The content of the RSA_BLOCK can be seen in Fig. 5.

\[ \text{msg3: } *\text{Stg} - K_{TPM}, \text{RSA_BLOCK}. \]

4. The RSA_BLOCK is decrypted by the pTPM and returned to the vTPMmgr, getting access to the sealed *Sig-KVM.

\[ \text{msg4: } vTPMData, \]

\[ H(vTPMData), \text{Stg} - K_{TPM}(\text{*Sig} - K_{VM}). \]
5. The vTPMmgr now gets access to $^{\ast}\text{Sig-}K_{\text{TPM}}$, sealed to the pTPM. It sends then a $\text{TPM2.Unseal()}$ command, in order to get access to the actual key handler.

\[ \text{msg5} : \text{Stg-K}_{\text{TPM}}(^{\ast}\text{Sig-K}_{\text{VM}}). \]

6. In case the PCRs are in the correct state, the pTPM can decrypt the key handler and return it to the vTPMmgr.

\[ \text{msg6} : ^{\ast}\text{Sig-K}_{\text{VM}}. \]

7. The vTPMmgr can now send the $\text{TPM2.Sign()}$ command to the TPM. The parameters of this command are $H(M)$ and $^{\ast}\text{Sig-K}_{\text{TPM}}$.

\[ \text{msg7} : H(M), ^{\ast}\text{Sig-K}_{\text{VM}}. \]

8. The signature is performed in the pTPM, encrypting $H(M)$ with $\text{Stg-K}^{-1}_{\text{TPM}}$. The result obtained, the signature is returned to the vTPMmgr.

\[ \text{msg8} : \text{Sig-K}_{\text{VM}}(H(M)). \]

9. The vTPMmgr sends then the signature to the vTPM instance.

\[ \text{msg9} : \text{Sig-K}_{\text{VM}}(H(M)). \]

10. Lastly, the vTPM sends it to the VM.

\[ \text{msg10} : \text{Sig-K}_{\text{VM}}(H(M)). \]

Listings for every message exchanged are provided in the Appendix A, as well as a brief description of the parameters.
Security considerations  Given a trusted hypervisor and a trusted vTPM-mgr, this approach achieves authentication of the VM and the physical system. VM authentication is provided since a VM can only ask the vTPM-mgr to decrypt its RSA_BLOk. The vTPM-mgr will only decrypt the vTPM data when the VM linked to that data needs it. At the same time, the vTPM data must be decrypted in order to get access to $\text{Sig-K}_{TPM}^{-1}$. A $\text{Sig-K}_{TPM}^{-1}$, stored in the pTPM, must be generated for every VM. Physical system authentication is provided since the actual private key is stored in the pTPM. In case the VM was migrated to another system by an attacker, the VM could never use the $\text{Sig-K}_{TPM}^{-1}$.

The fact that XEN Project hypervisor is an open-source hypervisor increases its trustworthiness, since the administrator of the system can read and analyse the code before compiling and installing it. Once the software has been installed, the trusted VMs are uploaded, the vTPM-mgr is instantiated and, a vTPM instance is generated for each of the VM needing a TPM. The system is at this point configured, ready to work and in a trusted state. The vTPM data can then be generated, and the $\text{Sig-K}_{TPM}^{-1}$s can be sealed to the system state. However, the PCR registers are only extended by default at start up and by the execution of the $\text{TPM}2\_\text{PCR}\_\text{Extend}()$ command. Consequently, code frequently updating the PCRs should be executed during the system operation. Thus, an illegitimate modification of the system would be detected and access to the vTPM data would not be possible.

In our procedure for signing messages, the vTPM-mgr has access to the plaintext $\text{Sig-K}_{TPM}^{-1}$. That key handler is used to reference the signature key. In a scenario with a malicious vTPM-mgr and/or hypervisor, the key handler could be used to sign any message from any malicious VM, impersonating the VM linked to that key. This could be possible as long as the VMs are running on the same system with the same TPM (the one which contains the private portion of the signature key). In case the vTPM-mgr or the hypervisor gets compromised, i.e. modified, it would be able to sign every message it wishes, impersonating the VM. However, after the $\text{TPM}2\_\text{PCR}\_\text{Extend}()$ command execution, the PCRs are updated with different values, reflecting the modification of the vTPM-mgr/hypervisor. Thus, the $\text{Sig-K}_{TPM}^{-1}$ handler could not be decrypted anymore. In spite of that, the plaintext key handler could have been stored and then used, even after a PCR modification.

Integrity of the system can be also verified by remote attestation. The remote attestation mechanism allows a challenger to see all relevant PCR measurements. PCR extensions are started early during the boot process, measuring firstly the BIOS, then the GRUB and the hypervisor. These measurements are stored in the lower PCRs of the pTPM. The upper PCRs store measurements of the vTPM-mgr and the VM’s operating system and applications. These upper PCRs, are VM-specific measurements, and are stored in the vTPM’s PCRs. The vTPM’s PCRs also store a copy of the lower PCRs stored in the pTPM.

A weak point of this approach is the link between the vTPM stub-domains and the VMs. Their communications are not authenticated. A compromised VM could take advantage of this and impersonate another VM and send as many digests as it wants to be signed. This could be possible until the first $\text{TPM}2\_\text{PCR}\_\text{Extend}()$ command execution happens after the attack against the VM.

We have considered in this approach that the hashing is performed by the
application willing to sign a message. However, that operation can also be performed in the TPM. This provides us with a better security level. When the application executes a $TPM2\_Hash()$ command, the hash and a ticket validating the digest are returned. The ticket indicates that the hashing has been performed in the pTPM. On the other hand, this would increment the overhead on the TPM and the number of times each VM needs to access the TPM.

4.4.2 Storing Signing Keys in the vTPMmgr

The vTPM implementation of XEN does not consider storing all the data related to vTPM within the pTPM. This approach would limit the number of vTPM instances running at a time as well as the performance. Rather, the vTPM data are stored within the vTPMmgr, encrypted with a pTPM key. Hence, when the application asks the TPM to sign a message, the communication between entities follows the same flow than in the previous approach (VM-vTPM-vTPMmgr-pTPM), however, in this case, the message to be signed is kept in the vTPM. Thereupon, the vTPM data are decrypted in the vTPMmgr and finally, the signature is performed within the vTPMmgr. The result is sent back to the vTPM and then to the VM.

This approach, although very similar than the previous one (storing the keys in the pTPM), is clearly less secure than the previous one. The vTPM keys travel between the different entities in plain-text. Therefore, the hypervisor as well as the process of creating the vTPMmgr and the vTPMs needs to be trusted, in order to extend the chain of trust rooted in the pTPM to VMs. However, it results in a performance improvement and in a scalable solution. We detail another approach also using this vTPM concept implemented in XEN, but in conjunction with TLS. This other approach, described in Section 5.1, was found to have more advantages, than the vTPM signature and therefore, vTPM signature was not further developed.

4.5 Intel® SGX

In this case, our scenario consists of a virtualised system, with several virtual machines running on it. The hardware of the system has an Intel® SGX-supported CPU. Every application running on every VM willing to sign messages has the capability of generate an enclave, which is only accessible to the application that generated it. We want to achieve VM and physical system authentication, in order to ensure that a received message is coming from legitimate software running on legitimate hardware. The basic crypto-operation that provides us authentication is the digital signature. We consider two approaches in order to achieve this: signing key generation within an enclave and, key provided by a third party.

4.5.1 Key Generation Within an Enclave

Since the Intel® architecture allows true random value generation by an enclave (RDRAND instruction), the public/private key pair can be generated within an enclave. When the application is launched, an enclave must be instantiated. Then, a signature key is generated inside the enclave. Obviously, the private
portion of the key would never leave the enclave, while the public portion would
be released. The signature operation would be performed within the enclave.
A certificate should be generated in order to bind the application/enclave
identity to its public key. This way, the receiving signed messages entity would
be able to check the origin of the message.

4.5.2 Key Provided by a Service Provider

According to the Intel® SGX specification, SGX-enabled software should not
be shipped with any sensitive data on it [16]. In our case, the sensitive data is
the private part of the signing key. This piece of sensitive data can be securely
 provisioned from a remote server. This can be performed taking advantage of
the attestation\(^5\) mechanism. Before the service provider provisions the keys to
the application, the service provider must ensure that it is sending the key to a
trusted environment. The application will provide to the third party information
about: what software is running inside the enclave, which execution environment
the enclave is running at, which sealing identity will be used by the enclave and
the CPU’s security level. A schema of the system and the communications
between entities can be seen in Fig. 7, as well as the schema of the messages
exchange between entities in Fig. 8. The procedure to achieve attestation and
secret transfer is explained below:

1. When the application is executed, it firstly instantiates an enclave. The
enclave’s measurements are calculated. The enclave contacts the service
provider (a remote server) in order to remotely retrieve data. The service
provider then replies with a challenge, since it must ensure that the appli-
cation is legitimate and actually running within an enclave. The challenge
contains a nonce to avoid replay attacks.

\(msg1: \text{Challenge, nonce.}\)

2. A special enclave, called quoting enclave is enabled. At this point the
application has two enclaves: the application’s enclave and the quoting
enclave. The application is provided with the quoting enclave’s MREN-
CLA VE.

\(msg2: \text{MRENCLA VE}\)

3. The application sends then the quoting enclave’s MRENCLA VE together
with the challenge to the application’s enclave.

\(msg3: \text{Challenge, Quoting enclave’s MRENCLA VE.}\)

4. The application’s enclave generates a response\(^6\) to the challenge and an
ephemeral private/public key pair. The response and the public key is
put together on a manifest. A hash digest of the manifest is generated.
The application’s enclave then invoke the EREPORT instruction that will

\(^5\)According to [16], Attestation in this context “is the process of demonstrating that a piece
of software has been properly instantiated on the platform”. Therefore, if the attestation has
been successful, a third party knows that the software is securely running within an enclave.
generate a REPORT binding the manifest to the enclave. The application’s enclave then sends the REPORT to the application.

msg4 : REPORT.

5. The application forwards it to the quoting enclave.

msg5 : REPORT.

6. The quoting enclave retrieves its Report Key from the CPU using EGETKEY. The REPORT is verified. At this point intra-platform attestation has been performed, the quote enclave knows that the application has instantiated an enclave. Within the quoting enclave, a QUOTE structure is created and signed with the EPID key. It is forwarded then to the application.

msg6 : QUOTE.

7. The application can send the QUOTE structure and the manifest to the service challenger. The challenger needs to verify the quote. In order to do so, it uses an EPID public key certificate to validate the signature over the quote. Then the integrity of the manifest is verified. At this point, the challenger verifies the content of the manifest searching for a valid response to the challenge sent in the first step.

msg7 : QUOTE, Manifest.

8. In the manifest, there is an ephemeral public key, which is used to encrypt the sensitive data to be transferred, in this case, the private part of the signing key bound to the application. The service provider sends then the encrypted signing key, which will be used during the application life.

msg8 : Secrets.

Once the attestation mechanism has finished successfully, the challenger has ensured that the software is legitimate and that it is running on the appropriate TCB (comparing the securely obtained MRENCLAVE and the expected one). This means that in case the VM is migrated to another system and/or the application gets compromised, the attestation would fail. In case of attestation fail, the application’s enclave would never receive the private signing key.

4.5.3 Observations

We have defined two approaches of providing a private key to an application’s enclave. Taking advantage of the attestation mechanism provided by Intel® SGX explained above, a signature key can be securely provided to an enclave using the second approach. Even though it seems to be secure enough, this

6Although in the documentation it is neither explained what the challenge consists in nor how the response is calculated, the authors claim it is a secure way to remotely transfer sensitive data to an enclave.
Figure 7: Schema of the involved entities during the SGX Inter-Platform attestation mechanism

Figure 8: Schema of messages exchange between entities during the SGX Inter-Platform attestation mechanism
approach of using private keys differs from the public/private key definition. Only the owner of the private key is supposed to store and have access to its private key. In the first approach the key is generated internally within an enclave, being therefore in accordance with the public key cryptography. Once the key has been provided to the enclave, both approaches behave the same way. The private signing key never leaves the enclave, and the signature procedure is performed within the enclave. A property of enclaves is that no software, not even privileged software, can read neither data nor code stored within an enclave. This means that there cannot be an attack able to compromise the private key once it is stored in an enclave.

In case an attacker with administrative access would duplicate the VM and boot the copied VM on the same system, the CPU would notice that is another piece of software, whose code is running on another memory space. Therefore, access to the original VM’s enclave is rejected. Moreover, if the duplicated VM and its enclave would be migrated to another system, with another CPU, it would never be able to access to the enclave’s data, since it is encrypted by a hardware-based encryption key. Further, private key stealing is also prevented. Only the piece of code running within the enclave can access to the plaintext key. Any other software running on the same VM could access, at most, to the hardware-encrypted key.

This approaches allow an easy software update procedure, ensuring the integrity and privacy of sensitive data. A persistent hardware-bound key is used to encrypt the enclave’s data. This data will return to its original state once the trusted environment of the upgraded software has been instantiated. The sealed data will no longer be available for previous versions of the software. Software updates might be provided by the service provider.
5 System Implementation

5.1 vTPM-TLS

Considering the advantages of the compatibility of TPM and TLS, we describe several ways of storing and managing TPM keys. In any case, these keys will be used during the TLS Handshake protocol in order to establish a secure session. As detailed in section 2.8 a TLS session provides both authentication and confidentiality. At the same time, TPM provides protection to the key storage.

5.1.1 Centralized TLS Key Distribution

In some cases, keys may need to be distributed in a centralized manner, i.e., one central IT server distributes all other entity’s keys. This can be specially interesting after a system upgrade or a system migration. In either case, the TPM may have to be replaced and it is an advantage being able to use the same key, and therefore, the same certificates. The TPM 2.0 design facilitates this feature.

As explained in section 2.3, when a TPM-provided system is set up, a SRK is generated on the system. The central system then keeps a record associating this key with the client entity ID. Prior to key distribution, the local entity gets the public portion of the central system’s key. Later, the following procedure takes place in order to distribute a key:

1. The central IT system generates a key, for instance, using TPM2_GetRandom().
2. The central IT system encrypts the key with the public portion of the client’s SRK.
3. The central IT system signs the encrypted key with its private signing key. This allows the local entity to know that what is being received is authorized by IT.
4. The encrypted key and the signature are sent to the client.
5. The client loads the server’s public key and verifies the signature on the encrypted key. This can be done with TPM2_Load() and then TPM2_VerifySignature().
6. The client imports the verified encrypted key into its TPM invoking TPM2_Import().
7. The client loads the key when it needs to use it using TPM2_Load().

See chapter 15 of [9] for further information on this and other centralized TPM key distribution approaches.

A similar approach could be applied in a scenario where the client entities do not have a TPM. The TPM-provided central server would distribute the key upon TLS session initialisation. The TLS HandShake protocol would use the key for performing the handshake and generating the session master secret. Later, the session between client and server can be resumed, avoiding the need of key redistribution every time the same secure connection needs to be performed. Provided that the session’s master secret has not been compromised, the session should be secure and independent from previous sessions.
However, the key would need to be in the local system’s memory. After session initialisation, it can be deleted and never used again, taking advantage of the TLS resuming sessions feature. Unfortunately this approach cannot be considered as secure as TPM-based ones. While the key is in system’s memory, malware could take it and use it later on illegitimate system. Moreover, in virtualised scenarios, without the TPM being involved in the client, the VM could be migrated by an attacker to another illegitimate machine, and the migrated (and illegitimate) VM would be able to use the central server-provided key.

5.1.2 vTPM-Stored Key

Taking advantage of the vTPM implementation for XEN hypervisor and GnuTLS, one is able to establish a TLS session between a VM and a TLS server. The physical machine needs to have a TPM attached to the motherboard. As explained in section 2.5.1, the vTPM’s data are stored encrypted with a pTPM key in the vTPMmgr stubdomain. For each 2048-bit RSA key generated with the vTPM, a certificate needs to be issued by a trusted certification authority. TPM key generation is straightforward using the GnuTLS-provided `tpmtool` command line utility. A UUID will be obtained after each TPM key generation.

GnuTLS also includes a tool to generate X.509 certificates and requests. The command line tool `Certtool` allows to generate our own privacy CA. It can be issue as many certificates as one needs signed by one’s privacy CA. Note that both TLS client and server need to trust the CA that signed the certificate bounding the vTPM key and the VM. However, the self-signed certificate of the privacy CA is not trusted by default.

The procedure to generate a privacy Certification Authority, an RSA key stored within the vTPM, a certificate bound to the key and lastly, establish a TLS session, is explained below. Note that GnuTLS needs to be compiled from source with PKCS11 and libtrousers.

First of all an RSA key to be stored in the TPM needs to be generated.

1 $ tpmtool --generate-rsa --bits 2048 --register --user

In case the SRK password is not the well-known value, will be prompted to enter the SRK password. The output of the command is the key UUID.

```
tpmkey: uuid=6c8374c8-b1d1-4567-82f9-c4a2d7146e8c;storage=user
```

Next, we extract the public part of the key we just generated. We will need it later.

2 $ tpmtool --pubkey "tpmkey:uuid=6c8374c8-b1d1-4567-82f9-c4a2d7146e8c;storage=user" --outfile=clientpublickey.pem

Hereafter, we generate a privacy Certification Authority. In order to do so, we first generate another RSA.

3 $ cerftool --generate--privkey --outfile MyCAKey.pem

An we are ready to generate the self-signed CA certificate

4 $ cerftool --generate--self--signed --outfile MyCA.pem --load--privkey MyCAKey.pem

Now that we have our own CA, we can generate as many certificates as we want. So we can generate the certificate bound to our key, that later will allow us to initiate TLS sessions.
We already have generated the key/certificate pairs for the client and the privacy Certification Authority. The only thing left is to generate it for the TLS server. The server could use a TPM-stored key too, like the client does. However, we will do it using our CA and certtool, storing the key in the file system with the following commands:

```
6 $ certtool --generate-cert --outfile servercert.pem
7 $ certtool --generate-cert --outfile serverkey.pem
```

At this point we have everything required to establish a TLS session. Our GnuTLS installation provided us with `gnutls-cli` and `gnutls-serv` command line tools. Taking advantage of it, we can set a local TLS server and a client that connects to it.

First, and using the above generated keys and certificates, we set up a TLS 1.2 Server in our local network.

```
8 $ gnutls-serv --x509-keyfile serverkey.pem --x509-certfile servercert.pem --x509-cafile MyCA.pem
```

The “-r” option will make the server require client authentication to reach a successful handshake. The rest of arguments just specify the required certificates and keys location within our file system.

If successful, we will get the following lines.

```
HTTP Server listening on IPv4 0.0.0.0 port 5556...done
HTTP Server listening on IPv6 :: port 5556...done
```

Once the server is running, we invoke the client tool to establish a secure session with it.

```
9 $ gnutls-cli --x509-keyfile "tpmkey:uuid=6c8374c8-b1d1-4567-82f9-c4a2d7146e8c;storage=user" --x509-certfile clientcert.pem
   --x509-cafile MyCA.pem -p5556 localhost
```

If we face some issues with the our CA and its certificate, we may solve it by using either `–tofu` (Trust On First Use) or `no-ca-verification` options. If successful, we will be prompted to introduce a PIN. This PIN is indeed the Storage Root Key we set when taking ownership of the TPM (or in our case, vTPM). We may avoid this by setting the environment variables `GNUTLS_PIN` and `GNUTLS_SO_PIN`.

---

On the client shell we should see the following information:

Processed 1 CA certificate(s).
Token 'SRK' with URL 'tpmkey:uuid=00000000-0000-0000-0000-000000000001;storage=user'
requires user PIN
Processed 1 client X.509 certificates...
Resolving 'localhost'...
- Certificate type: X.509
- Got a certificate list of 1 certificates.
- Certificate [0] info:
  
  Public Key ID:
  6754ea674200194db30ce61bf5e466dd4810af6

Public key's random art:

```
+---[ RSA 2048]----+
|                   |
|   === . oo       |
|  o + X + +      |
|    o @ o        |
|    o * E        |
|     . S = o     |
|      o +        |
```

- Status: The certificate is trusted.
- Successfully sent 1 certificate(s) to server.
- Description: (TLS1.2)–(ECDHE-RSA–SECP256R1)–(AES–128–GCM)
- Ephemeral EC Diffie–Hellman parameters
- Using curve: SECP256R1
- Curve size: 256 bits
- Version: TLS1.2
- Key Exchange: ECDHE-RSA
- Server Signature: RSA–SHA256
- Client Signature: RSA–SHA256
- Cipher: AES–128–GCM
- MAC: AEAD
- Compression: NULL
- Options: safe renegotiation
- Handshake was completed

Simple Client Mode:
On the client side, the following information should be displayed:

Processed 1 CA certificate(s).

* Accepted connection from IPv4 127.0.0.1 port 53123 on Sun Nov 8 16:00:20 2015
  - Status: The certificate is trusted.
  - Description: (TLS1.2) – (ECDHE-RSA-SECP256R1) – (AES-128-GCM)
  - Given server name [1]: localhost
  - Server has requested a certificate.
  - Certificate type: X.509
  - Got a certificate list of 1 certificates.
  - Certificate [0] info:
    - X.509 Certificate Information:
      - Version: 3
      - Serial Number (hex): 563f36dd2985a809
      - Issuer:
        - Not Before: Sun Nov 08 11:49:50 UTC 2015
        - Not After: Thu Apr 26 11:49:52 UTC 2018
      - Subject: CN=marcos
      - Subject Public Key Algorithm: RSA
      - Algorithm Security Level: Medium (2048 bits)
    - Modulus (bits 2048):
      - 3f
      - Exponent (bits 24):
        - 01:00:01
    - Extensions:
      - Basic Constraints (critical):
        - Certificate Authority (CA): FALSE
      - Key Purpose (not critical):
        - TLS WWW Client.
        - TLS WWW Server.
        - Ipsec IKE.
      - Key Usage (critical):
        - Digital signature.
        - Key encipherment.
      - Subject Key Identifier (not critical):
        - 039b5e3c9669e1e27184c64403c7358442d4675
      - Authority Key Identifier (not critical):
        - 06d7c658264de786fb240c735ce800aa6fa3cde0
      - Signature Algorithm: RSA-SHA256
      - Signature:
Another TPM-based option for storing keys would be to store them in the VM file system rather than within the vTPM. This can be achieved in two different ways:

- **Data binding**
- **Data sealing**

Data binding, in the TPM bibliography signifies that the data are encrypted using a TPM key, usually the SRK. Thus, even being easy to steal the data from the VM, it could not be read since it is encrypted. It can only be decrypted in the physical machine having the pTPM by the VM accessing to the appropriate vTPM. “Bound” keys could be generated very easily taking advantage of the GnuTLS tools too. Most of the commands explained in section 5.1.2 would apply for this keys too. The only different command would be command for generating the TPM key, the first command. If we decide to use bound keys instead of vTPM-stored keys, we would have to replace the above-mentioned command for the following one.

```bash
$ tpmtool --generate-rsa --sec-param=ultra --outfile=boundkey.pem
```

The “sec-param” parameter can be set to low, legacy, medium, high, ultra. Sec-param set to medium is equivalent to 2048-bit.

We can add an extra security layer to the generated keys by sealing them to the state of the system. This can be performed using the “tpm-tools” package, required to enable the TPM running a Linux system. The “tpm_sealdata” command line tool will only need an input file, an output file and the PCRs we want the data to be sealed to. First we need to generate a key (using for instance `certtool` or `OpenSSL`). That non-TPM-linked key can be sealed later, binding the key to the pTPM and to the system state. The following command can be executed to seal the “inputfile” to PCR 10, the default IMA PCR.

```bash
# tpm_sealdata -i inputfile -o outputfile -p 10
```

This way, “outputfile” is not only bound to our physical system but also to its “state”. Any software modification on the system, if IMA is enabled, is reflected on the IMA PCR and this would make it impossible to decrypt the sealed file.

Note that the sealed key would need to be unsealed prior usage. Once the key has been successfully unsealed, the procedure to initiate a TLS session would be very similar than the procedure explained in Section 5.1.2, but specifying the unsealed key rather than the vTPM key UUID.

This sealing option allows us to securely boot ParaVirtualised (PV) guests. Specifically in XEN, and using its nomenclature, the default for PV domains is for dom0 to pass the kernel to the domain builder. The dom0 administrator
boots PV guests by passing a guest configuration file to the hypervisor. This configuration file contains, among others, the path to the guest compiled kernel and the path to the guest filesystem. This solution is considered secure, but requires the administrator to keep up to date with kernel updates. It is not convenient for the guest user/administrator either, who can no longer choose its own kernel.

Xen provides two alternatives, both emulating GRUB for guests. The first alternative is to use *pygrub*. Pygrub is a python program that runs in domain 0. It reads the guest filesystem, finds the grub config files, interprets them to present a menu, then reads the selected file off the guest disk and passes it to the domain zero.

According to the official Xen documentation, this solution has one main drawback: “pygrub and the domain builder run in domain 0, the builder with full privileges to anything on the system. This makes them a juicy target for an attacker; any bug in python, the grub interpreter, the filesystem library, or the domain builder could be exploited to get full control of the system”.

The second alternative is *pvgrub*, a version of grub ported to run on minios. When using pvgrub, the domain builder will build the domain with the full amount of memory, but will start an image containing just minios and pvgrub. Pvgrub runs inside the new guest and will display the menu, and load and run the appropriate kernel. Since pvgrub runs inside the guest context, it has no more privileges than the guest already has. Therefore, there is no benefit to attacking it. Pvgrub can be run by simply setting the kernel in the guest configuration file to point to the appropriate pvgrub image rather than the kernel itself.

By default, the vTPM starts up with all PCRs set to the well-known value (all zeros for the lower 16 PCRs). This means that any decision about the trustworthiness of the created domain must be made based on the environment that created the vTPM and the domU. For example, a system that only constructs images using a trusted configuration and trusted guest kernel can provide guarantees about the guests and any measurements done that kernel (such as the IMA TCB log). However, guests wishing to use a custom kernel in such a secure environment are often started using the pvgrub bootloader as the kernel, which then can load the untrusted kernel without needing to parse an untrusted filesystem and kernel from dom0. If the pvgrub stub domain succeeds in connecting to its vTPM, it will extend the hash of the kernel that it boots into PCR 4, and will extend the command line and initrd into PCR 5 before booting so that a domU booted in this way can attest to its early boot state.

Allowing a vTPM to remain active across guest restart will cause the PCR values to be extended by pv-grub, and they will be incorrect. If the domU had not had any modification at all, the measurement will be the same, however, this measurement will extend the previous one, making a different final value. In order for the vTPM’s PCRs to be useful for quotes or releasing sealed secrets, you need to ensure that a new vTPM is started if and only if it is paired with a corresponding guest.

Unlike is claimed in the Xen documentation, we have experienced that this does not always occur. We have managed to boot a domU and it successfully connected to its vTPM (it was completely functional) and PCRs were still set to the well-known value. Nevertheless, we can still use PCR-10, IMA PCR, to provide attestation of the VM’s state, detecting any software modification/up-

41
date. It is explained in section 5.1.3 how to take advantage of PCRs in order to
seal key to the state of the machine.

5.1.4 Integration with EST

Libest8 is a publicly available EST implementation for Linux. It contains a basic
implementation as well as server and client examples. The examples provided
with the implementation allow us to ask the EST server for keys or use our own
keys. Unfortunately, the examples are not compatible with TPM UUIDs and therefore, we can only use keys generated with certtool and stored in the file
system, but not TPM keys. Although it was not meant to be compatible with
TPM and GnuTLS, we were able to find a workaround that allows us to ask an
EST server to re-issue certificates bound to a private key stored in a TPM (or
vTPM).

Since certtool is compatible with TPM keys, we may generate a Certificate
Signing Request (CSR). This certificate signing request includes the public key
along with information about its owner’s identity. In our case, it should include
information about the VM and the physical machine it is running on. The
CSR contains a signature, proving possession of the private key. Prior to CSR
generation, it is needed to be in possession of the vTPM key, as explained in
section 5.1.2. Once we know the vTPM-key UUID, we may invoke the following
command:

```
$ certtool --generate-request --load-privkey "tpmkey:uuid=340b87c2
-c43d-4986-8a82-3dc6c4ce0593;storage=user" --load-pubkey
pubkey.pem --outfile request.pem
```

Thereupon, once libest has been installed, we may use this CSR to request
the certificate itself to the EST server.

```
$ ./estclient -e -s 127.0.0.1 -p 8085 -u estuser -h estpwd
-y req.pem -o /tmp --pem-output
```

Note that in the EST Server example provided with libest, the HTTP user
and password are hard-coded and are “estuser” and “estpwd” respectively. The
certificate could be issued by the EST Server by only providing the HTTP user
and password and the private key, however, the EST implementation would
have to be extended in order to be compatible with PKCS#11, specifically with
TPM-keys UUID.

---

8Available at https://github.com/cisco/libest
6 Performance Testing

We have measured the delay introduced by calling the TPM and the vTPM for performing the TLS Handshake. They are compared versus the TLS handshake performed using 2048-bit RSA keys plain-text stored in the file system. All the measurements have been taken with the following parameters:

- Ephemeral Elliptic Curve Diffie-Hellman, with 256-bit curve size.
- TLS version 1.2.
- Ephemeral Elliptic Curve Diffie-Hellman-RSA key exchange.
- RSA-SHA256 Server and Client signatures.
- AES-128-GCM cipher.

The measurements have been taken using two computers (one running the TLS Server and the other one running the TLS Client). Both computers are within the same Local Area Network. Each scenario has been measured five times in total, with deviations of few milliseconds. Each measurement considers the time elapsed from the instant the first TCP segment with the SYN flag set is received by the Server, until the first encrypted application data is received by the Server after the handshake.

We show in Table 1 all the measured times for the four considered scenarios. The average measurement is shown in Figure 9 as a bar graph.

<table>
<thead>
<tr>
<th>Iteration</th>
<th>File System-stored</th>
<th>VM FS-stored</th>
<th>TPM</th>
<th>vTPM</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.059717</td>
<td>0.058996</td>
<td>0.304315</td>
<td>0.077479</td>
</tr>
<tr>
<td>2</td>
<td>0.052106</td>
<td>0.065163</td>
<td>0.304328</td>
<td>0.073058</td>
</tr>
<tr>
<td>3</td>
<td>0.056508</td>
<td>0.061383</td>
<td>0.302823</td>
<td>0.077150</td>
</tr>
<tr>
<td>4</td>
<td>0.057028</td>
<td>0.060855</td>
<td>0.300357</td>
<td>0.074240</td>
</tr>
<tr>
<td>5</td>
<td>0.056728</td>
<td>0.0617765</td>
<td>0.304350</td>
<td>0.077970</td>
</tr>
<tr>
<td>Average</td>
<td>0.056</td>
<td>0.061</td>
<td>0.303</td>
<td>0.075</td>
</tr>
</tbody>
</table>

Table 1: TLS Handshake time (seconds) elapsed for each iteration for each scenario.

Note that mutual authentication is considered in the four different solutions compared. The server is always performing the handshake with a key stored in its file system. Both file system-stored key solutions store the keys in plain-text, i.e., not encrypted. Therefore, these solutions are expected to be more efficient that the rest. Referring to the results, we see that the non-virtualised file system-stored solution offers the best results, followed by the virtualised solution. These results are what we expected. File system-stored key should be the fastest solution, and running the same OS over the hypervisor is expected to add a slight delay.

Comparing the TPM-based solutions, we find that the same approach (invoking exactly the same commands for establishing the TLS session) the virtualised scenario is notably more efficient than the non-virtualised one. This happens because every cryptographic operation in the non-virtualised scenario
during the handshake is performed directly in the pTPM. Nevertheless, in the virtualised scenario the pTPM is only used once to decrypt the vTPM data, and then, software entities perform the operations.
7 Discussion

Comparing every approach explained above we realise that “VM Certification” as well as “Code Signing” approaches have the advantage that they can be implemented on any system. They do not need any specific hardware, unlike all other approaches. However, neither of the approaches fulfil requirement R4 System (software and hardware) trustworthiness. Hardware trustworthiness is not considered at all on these approaches. Thus, these approaches could only be useful in scenarios where requirement R4 is not actually a requirement.

Next detailed approach, “Signature Chain” is the most straightforward approach keeping in mind R4 needs to be achieved. Although R4 is achieved, this approach requires two signatures of every message, compromising R3 Low-Latency. Further, this approach is not directly compatible with any already written code. Every application already signing messages that are meant to be used in this approach would have to be re-written, considering both signatures.

Regarding the SGX-based solution, at the time of writing this report, SGX has not been released yet. Due to this reason, we can not actually test it. However, it seems to be the most complete and efficient approach: we do not need to trust any software running on the CPU, not even the hypervisor. Every cryptographic operation is performed in the CPU, which is much faster than the TPM. The main drawback of this approach is that it can only be implemented on a system provided with an Intel® SGX CPU.

Both approaches involving vTPM Signature (described in section 4.4) are valid. However, we found the approach involving vTPM and TLS more appropriate. There is a lot of software written using TLS. This approach makes the vTPM use transparent, making a TLS client using this vTPM approach able to connect to any TLS server, not necessarily having a TPM. Moreover, by using the vTPM Signing Keys in the pTPM approach, the hypervisor (and its vTPM implementation) would have to be extended/modified.

For these reasons, we decided to develop our solution with vTPM, TLS and Xen hypervisor. Other approaches or solutions described in this report either do not fulfil our requirements or are infeasible. Many other hypervisors are available on the market, however, they are either not open-source or do not provide any feature for achieving virtual machine authentication.
8 Conclusions and Future Work

8.1 Conclusions

With the goal of providing authentication to virtualised applications, we designed several approaches that may be split into two groups: Software-stored SecDevID and Hardware-rooted SecDevID. On one hand, software-based solutions provide good performance and can be implemented on every system, with no hardware limitations. On the other hand, hardware-based solutions bind the software to the hardware, assuring us our software is only running on one hardware machine.

We implemented the hardware-based approach that best met the requirements presented in Section 3, i.e., vTPM-TLS. This approach provides trusted computing to every virtual machine, as well as TLS connectivity, allowing a TLS server to trust the TLS client’s software and hardware. At the same time, as shown in Section 6, this solution does not significantly compromise the performance in terms of delay. Furthermore, not only authentication but also confidentiality and integrity is achieved.

8.2 Future Work

At the time of developing this work, the last version of Xen Hypervisor (4.5) was only TPM1.2-compilant. However, the next stable version (4.6) has already been released, which is compatible with TPM2.0 standard. TPM2.0 provides several new features and improvements, that could be applied to the approaches presented in this report (such as Elliptic Curve Cryptography, a more secure and more efficient public-key cryptography approach than RSA).

In the presented approaches, no mention to IEEE 802.1X is made. The approaches detail, in general, a way of storing secrets and using them. This is highly related to IEEE 802.1AR. Further work could be done extending the presented approaches to be compliant with IEEE 802.1X and make the network access control the objective of the approaches, instead of the authentication of devices (or virtual machines).

None of the detailed approaches in this report achieves hypervisor authentication. Therefore, the hypervisor needs to be trusted. The Shielding applications concept, detailed in [28], was developed considering the fact that in untrusted clouds environments, the hypervisor cannot be trusted. Further research can be done following this idea, and SGX seems to be a good technology to provide security to our applications running on untrusted clouds.
References


A  vTPM code

Listing 1: Messages 1 7 8 and 10

<table>
<thead>
<tr>
<th>IN PARAMETERS</th>
<th>Description</th>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM2_B_COMMAND_HEADER</td>
<td></td>
<td>UINT32</td>
<td>tag</td>
<td>TPM_ST_SESSIONS</td>
</tr>
<tr>
<td>TPM_CC</td>
<td></td>
<td>UINT32</td>
<td>commandSize</td>
<td>TPM_CC_Sign</td>
</tr>
<tr>
<td>TPM_DH_OBJECT</td>
<td></td>
<td>UINT32</td>
<td>commandCode</td>
<td>TPM_ALG_NULL</td>
</tr>
<tr>
<td>TPM2B_DIGEST</td>
<td></td>
<td>UINT32</td>
<td>keyHandle</td>
<td>TPM_ALG_NULL</td>
</tr>
<tr>
<td>TPMT_SIG_SCHEME+</td>
<td>Digest to be signed.</td>
<td>UINT32</td>
<td>digest</td>
<td>Digest to be signed.</td>
</tr>
<tr>
<td></td>
<td>Signing scheme to use if the scheme for keyHandle is TPM_ALG_NULL.</td>
<td></td>
<td>inScheme</td>
<td>Signing scheme to use if the scheme for keyHandle is TPM_ALG_NULL.</td>
</tr>
<tr>
<td>TPMT_TK_HASHCHECK</td>
<td>Proof that digest was created by the TPM. In our case, this parameter is NULL.</td>
<td></td>
<td>validation</td>
<td>Proof that digest was created by the TPM. In our case, this parameter is NULL.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OUT PARAMETERS</th>
<th>Description</th>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_ST_TAG</td>
<td></td>
<td>TPM_ST_SESSIONS</td>
<td>tag</td>
<td>TPM_ST_SESSIONS</td>
</tr>
<tr>
<td>UINT32</td>
<td></td>
<td>TPM_RC</td>
<td>responseSize</td>
<td>TPM_ALG_NULL</td>
</tr>
<tr>
<td>UINT32</td>
<td></td>
<td>TPM_RC</td>
<td>responseCode</td>
<td>TPM_ALG_NULL</td>
</tr>
<tr>
<td>TPMT_SIGNATURE</td>
<td>The signature.</td>
<td>TPMT_SIGNATURE</td>
<td>signature</td>
<td>The signature.</td>
</tr>
</tbody>
</table>

Listing 2: Message 2

```c
/* Create the command */
len = size = VTPM_COMMAND_HEADER_SIZE;
bptr = cmdbuf = malloc(size);
TRYFAILGOTO(pack_header(&bptr, &len, tag, size, ord));

/* Send the command to vtpm_manager */
info("Sending Sign command to backend");
TRYFAILGOTOMSG(tpmfront_cmd(tpmfront_dev, cmdbuf, size, &resp, &resplen), ERR_TPMFRONT);
```

9When the command is executed at message 1, the key handler is irrelevant. It might be a random handler. Nevertheless, during the execution of the command at message 7, we already know the proper key handler, and this parameter must be passed correctly in order to sign using the appropriate key.
Listing 3: Messages 3 and 4

```
// IN: input parameter list
// OUT: output parameter list
TPM_RC TPM2_RSA_Decrypt(RSA_Decrypt_In *in, RSA_Decrypt_Out *out)
```

The cipher text to be decrypted in this operation is the whole `RSA_BLOCK` associated to the vTPM, stored in the vTPMmgr.
### IN PARAMETERS

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPML_ST COMMAND_TAG</td>
<td>tag</td>
<td>TPM_ST_SESSIONS</td>
</tr>
<tr>
<td>UINT32</td>
<td>commandSize</td>
<td></td>
</tr>
<tr>
<td>TPM_CC</td>
<td>commandCode</td>
<td>TPM_CC_RSA_Decrypt</td>
</tr>
<tr>
<td>TPML_DH_OBJECT</td>
<td>keyHandle</td>
<td>RSA key to use for decryption. In our case, a TPM storage key.</td>
</tr>
<tr>
<td>TPM2B_PUBLIC_KEY_RSA</td>
<td>cipherText</td>
<td>Cipher text to be decrypted. An encrypted RSA data block is the size of the public modulus.</td>
</tr>
<tr>
<td>TPMT_RSA_DECRYPT+</td>
<td>inScheme</td>
<td>The padding scheme to use if scheme associated with keyHandle is TPM_ALG_NULL.</td>
</tr>
<tr>
<td>TPM2B_DATA</td>
<td>label</td>
<td>Label whose association with the message is to be verified.</td>
</tr>
</tbody>
</table>

### OUT PARAMETERS

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_ST</td>
<td>tag</td>
<td></td>
</tr>
<tr>
<td>UINT32</td>
<td>responseSize</td>
<td></td>
</tr>
<tr>
<td>TPM_RC</td>
<td>responseCode</td>
<td></td>
</tr>
<tr>
<td>TPM2B_PUBLIC_KEY_RSA</td>
<td>message</td>
<td>Decrypted output.</td>
</tr>
</tbody>
</table>

Listing 4: Message 5 and 6

```c
// IN: input parameter list
// OUT: output parameter list
TPM_RC TPM2_Unseal( Unseal_In *in, Unseal_Out *out )
```
### IN PARAMETERS

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPML_ST_COMMAND_TAG</td>
<td>tag</td>
<td>TPM_ST_SESSIONS</td>
</tr>
<tr>
<td>UINT32</td>
<td>commandSize</td>
<td></td>
</tr>
<tr>
<td>TPM_CC</td>
<td>commandCode</td>
<td>TPM_CC_Unseal</td>
</tr>
<tr>
<td>TPML_DH_OBJECT</td>
<td>itemHandle</td>
<td>Handle of a loaded data object.</td>
</tr>
</tbody>
</table>

### OUT PARAMETERS

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_ST</td>
<td>tag</td>
<td></td>
</tr>
<tr>
<td>UINT32</td>
<td>responseSize</td>
<td></td>
</tr>
<tr>
<td>TPM_RC</td>
<td>responseCode</td>
<td></td>
</tr>
<tr>
<td>TPM2B_SENSITIVE_DATA</td>
<td>outData</td>
<td>Unsealed data. Size of outData is limited to be no more than 128 octets.</td>
</tr>
</tbody>
</table>

Listing 5: Message 9

```c
/* Unpack response header */
bptr = resp;
len = resplen;
TRYFAILGOTOMSG(unpack_header(&bptr, &len, &tag, &size, &ord), ERR_MALFORMED);
```
B Intel® SGX’s data structures

The REPORTDATA structure specifies the address of a 64-Byte input buffer that the EREPORT instruction will use to generate cryptographic report. It requires 128-Byte alignment.

The TARGETINFO structure is an input parameter to the EREPORT instruction leaf. The address of TARGETINFO is specified as an effective address in RBX. It is used to identify the enclave which will be able to cryptographically verify the REPORT structure returned by EREPORT. A TARGETINFO requires 128-Byte alignment.

<table>
<thead>
<tr>
<th>Field</th>
<th>Offset (bytes)</th>
<th>Size (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEASUREMENT</td>
<td>0</td>
<td>32</td>
<td>The MRENCLAVE of the target enclave</td>
</tr>
<tr>
<td>ATTRIBUTES</td>
<td>32</td>
<td>16</td>
<td>The ATTRIBUTES field of the target enclave</td>
</tr>
</tbody>
</table>
The REPORT structure is the output of the EREPORT instruction, and must be 512-Byte aligned.

**REPORT structure**

<table>
<thead>
<tr>
<th>Field</th>
<th>Offset (bytes)</th>
<th>Size (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPUSVN</td>
<td>0</td>
<td>16</td>
<td>The security version number of the processor.</td>
</tr>
<tr>
<td>RESERVED</td>
<td>16</td>
<td>32</td>
<td>Must be zero</td>
</tr>
<tr>
<td>ATTRIBUTES</td>
<td>48</td>
<td>16</td>
<td>The values of the attributes flags for the enclave.</td>
</tr>
<tr>
<td>MRENCLAVE</td>
<td>64</td>
<td>32</td>
<td>The value of SECS.MRENCLAVE</td>
</tr>
<tr>
<td>RESERVED</td>
<td>96</td>
<td>32</td>
<td>Reserved</td>
</tr>
<tr>
<td>MRSIGNER</td>
<td>128</td>
<td>32</td>
<td>The value of SECS.MRSIGNER</td>
</tr>
<tr>
<td>RESERVED</td>
<td>160</td>
<td>96</td>
<td>Zero</td>
</tr>
<tr>
<td>ISVPRODID</td>
<td>256</td>
<td>02</td>
<td>Enclave PRODUCT ID</td>
</tr>
<tr>
<td>ISVSVN</td>
<td>258</td>
<td>02</td>
<td>The security version number of the Enclave</td>
</tr>
<tr>
<td>RESERVED</td>
<td>260</td>
<td>60</td>
<td>Zero</td>
</tr>
<tr>
<td>REPORTDATA</td>
<td>320</td>
<td>64</td>
<td>A set of data used for communication between the enclave and the target enclave. This value is provided by the EREPORT call in RCX.</td>
</tr>
<tr>
<td>KEYID</td>
<td>384</td>
<td>32</td>
<td>Value for key wear-out protection.</td>
</tr>
<tr>
<td>MAC</td>
<td>416</td>
<td>16</td>
<td>The CMAC on the report using report key.</td>
</tr>
</tbody>
</table>

*Offset and Size fields measured in bytes.*