Master thesis

On the security of Virtual Machine migration and related topics

Ramya Jayaram Masti
Submitted in fulfillment of the requirements of
Master of Science in Computer Science
Department of Computer Science

Advisors:
Boris Dimitrov Danev
Ghassan O. Karame
Prof. Dr. Srdjan Ćapkun

April 2010 - October 2010

System Security Group
Department of Computer Science
ETH Zurich
Abstract

The extension of trusted computing to virtual environments in order to provide secure storage and ensure system integrity presents several challenges. In particular, techniques to share a hardware TPM between several virtual machines (VM) on the same host and to appropriately secure these VMs during migration need to be devised. In this work, we provide a comprehensive overview of trusted computing and its extension to virtualization by means of virtual TPMS (vTPM). We analyze existing vTPM designs and related VM-vTPM migration protocols and derive a set of requirements for the secure VM-vTPM migration. We then propose a secure migration protocol using a novel vTPM key hierarchy that satisfies these requirements. We implement our protocol and evaluate its performance using different ciphers and VM RAM sizes. Our results show that both RC4 and 128-bit AES are efficient as underlying ciphers as opposed to 3DES which introduces a significant overhead on the migration process. Moreover, the end-to-end migration time experienced by the end user (using RC4 or 128-bit AES as the underlying cipher) increases only by approximately 10% compared to the insecure migration when VM RAM size is larger than 512 MB. This overhead may be tolerable in certain applications depending on the used hardware. Thus, secure VM-vTPM migration can be practical.
Acknowledgements

On the completion of my master thesis, I would like to express my gratitude to everyone who has helped me through the course of this project. At the outset, I would like to thank Prof. Dr. Srdjan Čapkun for giving me the opportunity to work in his group. It has helped me experience all phases of a research project in preparation for my doctoral studies. I am grateful to him for his guidance and encouragement during this period.

I am very fortunate to have worked under the guidance of Ghassan O. Karame and Boris Dimitrov Danev. I thank Ghassan for his motivation and support through all phases of the work. I appreciate his patience and understanding in putting up with my many questions despite his busy schedule. This work would have not been possible without the help of Boris from whom I have learned a lot technically and professionally. His advice, patience and encouragement have been invaluable to me and I thank him for that.

Special thanks are due to Dr. Mario Strasser for his assistance regarding the TPM emulator, Kasper Bonne Rasmussen for this help in Linux related matters and Nathalie Weiler from Credit Suisse for an industry perspective on cloud computing. I thank my friends in the student room for the excellent company and good conversations. Last but not the least, I thank my family for their unflinching support during this project.
Contents

1 Introduction  

2 Background  3
   2.1 Virtualization  ........................................... 3
      2.1.1 Security implications of virtualization  .......... 4
   2.2 Trusted computing  ...................................... 4
      2.2.1 Trusted Platform Module  .......................... 4
      2.2.2 Roots of Trust  ..................................... 4
      2.2.3 Components of a TPM  ............................... 5
      2.2.4 TPM keys  .......................................... 5
      2.2.5 The TCG Software Stack (TSS)  .................... 6
      2.2.6 TPM applications  .................................. 6
      2.2.7 Relay and Cuckoo attacks  .......................... 9
      2.2.8 Issues with trusted computing  ..................... 10
   2.3 TPM virtualization  ..................................... 10
      2.3.1 Software virtualization of TPMs: by Berger S, et al., 2006  . 11
      2.3.2 Hardware virtualization of TPMs: by Stumpf F, et al., 2008  . 12
      2.3.3 Para-virtualized TPM sharing: by England P, et al., 2008 . 13
      2.3.4 Property-based TPM virtualization: by Sadeghi A, et al., 2008  . 15
      2.3.5 TPM Virtualization: Building a General Framework: by Scar-
lata V, et al., 2007  ........................................... 16
      2.3.6 The TPM emulator  .................................... 17

3 Overview and analysis of existing VM-vTPM migration protocols 19
   3.1 Motivation for secure VM-vTPM migration  ............... 19
   3.2 Xen’s existing VM migration protocol  .................... 19
   3.3 Review of the existing vTPM migration protocols  ....... 20
      3.3.1 Protocol 1: by Berger S, et al., 2006  ................ 20
      3.3.2 Protocol 2: By Stumpf F, et al., 2008  ............... 21
      3.3.3 Protocol 3: inferred from the source code of the Xen hypervisor  .... 21
      3.3.4 Protocol 4: by Sadeghi A, et al., 2008  ................ 22
   3.4 Evaluation of the proposed protocols  ................... 23
      3.4.1 The lack of a context  ................................ 23
      3.4.2 Secure VM-vTPM association  ........................ 24
      3.4.3 Authenticated transfer to a secure destination platform  .......... 26
3.4.4 Confidentiality and integrity preserving and replay resistant transfer ........................................... 26
3.4.5 Atomicity of the migration .............................................................................................................. 27
3.4.6 vTPM key hierarchies: Validity and implications for migration ......................................................... 28
3.4.7 Privacy preserving migration ......................................................................................................... 30
3.5 Requirements for a migration protocol ............................................................................................... 32

4 Migration Protocol Design ....................................................................................................................... 35
4.1 vTPM key hierarchy design ................................................................................................................. 35
  4.1.1 vTPM keys .................................................................................................................................... 35
  4.1.2 Keys in a suspended VM’s vTPM ................................................................................................. 37
  4.1.3 vTPM key transfer during migration ............................................................................................ 38
4.2 A note on the integrity protection of vTPM keys .................................................................................. 39
4.3 Protocol Outline .................................................................................................................................... 39
4.4 Detailed design ...................................................................................................................................... 41
  4.4.1 Phase I: Secure session establishment ......................................................................................... 41
  4.4.2 Phase II: Remote attestation of the destination ............................................................................... 43
  4.4.3 Phase III: vTPM and VM transfer ............................................................................................... 43
  4.4.4 Phase IV: Deletion at the source .................................................................................................... 44
  4.4.5 A note on VM attestation .............................................................................................................. 44
4.5 Protocol evaluation .................................................................................................................................. 44

5 Implementation ......................................................................................................................................... 47
5.1 The implementation choices .................................................................................................................. 47
  5.1.1 The hypervisor ............................................................................................................................. 47
  5.1.2 The vTPM architecture ................................................................................................................ 48
5.2 The implementation setup ...................................................................................................................... 49
  5.2.1 The hardware ................................................................................................................................ 49
  5.2.2 Xen installation .............................................................................................................................. 49
  5.2.3 The hardware TPM setup ............................................................................................................. 50
  5.2.4 The TPM emulator (vTPM) setup ................................................................................................. 50
  5.2.5 Trusted boot .................................................................................................................................. 51
  5.2.6 The shared storage setup ............................................................................................................. 52
5.3 The implemented insecure protocol ...................................................................................................... 52
  5.3.1 VM suspension and restoration .................................................................................................... 52
  5.3.2 VM migration with or without disk transfer .................................................................................. 53
5.4 The implemented secure migration protocol .......................................................................................... 53
  5.4.1 Secure connection setup .............................................................................................................. 53
  5.4.2 Attestation ..................................................................................................................................... 55
  5.4.3 File locking .................................................................................................................................... 56
5.5 Known issues .......................................................................................................................................... 56

6 Performance evaluation .............................................................................................................................. 57
6.1 Methodology .......................................................................................................................................... 57
  6.1.1 Performance metrics ...................................................................................................................... 57
  6.1.2 Measured protocol operations ..................................................................................................... 58
6.1.3 Measurement technique ........................................ 59
6.2 Analysis of results .................................................. 61
  6.2.1 Connection establishment ...................................... 61
  6.2.2 VM suspension and resumption time .......................... 62
  6.2.3 End-to-end migration .......................................... 62
6.3 Summary of evaluation results .................................... 68

7 Conclusion and outlook ............................................. 71
Chapter 1

Introduction

Cloud computing is the delivery of computing resources and applications as a service over the Internet. A 'cloud provider' provides resources (e.g. infrastructure, platform and/or software) to a 'cloud user' on demand. A key technological enabler of cloud computing is virtualization. Virtualization enables the abstraction of physical resources for the purpose of sharing it between multiple clients. This improves hardware utilization leading to reduced operational and investment costs. Besides these benefits, some features of virtualization are beneficial to security. These include increased isolation and increased ability for intrusion detection and prevention [1]. However, virtualization also creates new security challenges. The ease of duplication of Virtual Machines (VMs) makes identity management a challenge. The diversity and mobility of VMs complicates software life-cycle management [2]. For example, protecting sensitive information from unauthorized access and ensuring the integrity of software components are challenging tasks. Currently, (in)security of virtual systems is being researched widely [3], [4].

The above security concerns and related requirements have led to the incorporation of trusted computing into virtualized systems. Trusted computing and Trusted Platform Modules (TPMs) facilitate secure storage of sensitive information and allow verification of system integrity. While the application of trusted computing to securing virtual systems is a natural extension to its original scope, its realization poses several challenges. This extension of TPMs for use with VMs is called TPM virtualization and it results in a virtual TPM (vTPM) design.

vTPMs facilitate sharing of the hardware TPM between several VMs. But due to infrastructure management, change in policies and dynamic load balancing, VMs are often forced to move between different hardware platforms. This process is referred to as VM migration. The VMs can either be stopped or suspended before migration or migrated 'live' in order to minimize downtime. In the context of trusted computing enabled systems, VM migration requires the transfer of the vTPM along with the VM in order to ensure normal operation of applications that use the vTPM. Further, it is important to secure VM-vTPM migration because VM migration over the Internet is vulnerable to all the threats characterizing data exchange over a public network. These include leakage, modification and loss of sensitive information.
The complexity of vTPM migration depends upon the vTPM architecture in use. A vTPM could run in a separate VM, inside its corresponding VM or in the hypervisor. Its relationship with the hardware TPM could also vary depending upon its design. The possibility of adding of vTPM migration to suspended VM migration and live VM migration has to be analyzed. In addition to the security requirements of VM migration, a vTPM migration protocol has to address several other issues. These include timing of vTPM migration with respect to VM migration and ensuring the availability of data that is protected using the vTPM at the destination. To ensure the availability of data, one requires a vTPM key hierarchy that complies with TPM key hierarchy semantics and facilitates migration.

There have been attempts in the recent years to design and implement vTPM architectures and vTPM migration protocols. In this work, we survey existing solutions to VM-vTPM architecture design and vTPM migration. We derive a set of requirements for secure vTPM and VM migration by analyzing the security properties of these existing solutions. We then design and implement a secure VM-vTPM migration protocol with a novel vTPM key hierarchy. Finally, we evaluate the performance of our protocol with respect to the standard insecure VM-vTPM migration protocol.

The rest of this report is structured as follows. First, we briefly review trusted computing technologies and existing vTPM design proposals in Chapter 2. In Chapter 3, we discuss the motivation for secure VM-vTPM migration and analyze the security properties of existing vTPM migration protocols. In Chapter 4, we present a new key hierarchy for vTPMs and propose a new VM-vTPM migration protocol. The implementation of our secure migration protocol and its performance evaluation are described in Chapter 5 and Chapter 6 respectively. Finally, in Chapter 7, the conclusions of our study and scope of future work is presented.
Chapter 2

Background

In this chapter, an overview of virtualization and its security implications, trusted computing and TPM virtualization solutions is provided.

2.1 Virtualization

Virtualization is the abstraction of hardware resources to enable improved sharing between multiple clients. Different levels of virtualization have been identified including, hardware virtualization, operating system virtualization and application virtualization [5]. In [6], the authors present a detailed comparison of software and hardware virtualization techniques and argue that software virtualization results in better performance than hardware virtualization.

Virtual Machine Monitors (VMMs) and the Virtual Machines (VMs) are two central entities in a virtualized environment. They were first defined by Popek and Goldberg in [7]. A VM was defined as ‘an efficient isolated duplicate of a real machine’. They defined a VMM as a layer of software that satisfied the following three properties:

a. A VMM provides VMs with an environment that is ”essentially identical” to the real machine.

b. The performance of programs degrades very little when they execute in a VM rather than on the real machine.

c. The VMM completely controls system resources.

Today, VMMs can be broadly classified into two categories. Type I VMMs (also called hypervisors) run on the bare hardware while Type II VMMs run on the host OS as described in [8]. Several commercial and open-source hypervisors are available today including VMware’s vSphere Hypervisor (ESXi) (http://www.vmware.com/) and the Xen hypervisor (http://www.xen.org/).
2.1.1 Security implications of virtualization

Virtualization facilitates better sharing of resources which leads to savings in investment and operational costs. It has many security implications too. The increased isolation helps to contain system compromise and the ease of creating and destroying VMs gives increased ability for intrusion detection and prevention. But identity management, access control, secure storage and ensuring system integrity in virtualized systems is non-trivial. Trusted computing is a technology that can be used to partially address these security concerns.

2.2 Trusted computing

Trusted computing is an approach to building systems such that their integrity can be verified. It is based on the concept of transitive trust where initial trust in a h/w module is delegated to other system components. The initially trusted h/w module is called a Trusted Platform Module (TPM).

2.2.1 Trusted Platform Module

We briefly review the TCG consortium’s specifications for TPMs (available at http://www.trustedcomputinggroup.org/developers/trusted_platform_module). In particular, we discuss the components of a TPM, the TPM Software Stack (TSS) and key generation in the TPM framework as described in the specification documents. Finally, we describe some privacy concerns regarding the adoption of trusted computing.

2.2.2 Roots of Trust

There are certain parts of the TPM that have to be trusted in order to bootstrap trust in other parts of the system [9]. There are three such roots of trust and are described below.

a. Root of Trust for Measurement (RTM): This is a computing engine capable of making reliable measurements.

b. Root of Trust for Storage (RTS): This is a computing engine capable of reliably storing integrity digests and sequences of digests.

c. Root of Trust for Reporting (RTR): This is a computing engine capable of reliably reporting the information in the RTS.

Finally, we distinguish between a static root of trust and a dynamic root of trust. A static root of trust begins from a well known starting state such as a power-on-self-test. A dynamic root of trust transitions from an untrusted state to one that is trusted.
2.2.3 Components of a TPM

Figure 2.1 shows the components of a TPM. The TPM consists of number of cryptographic modules including a random number generator, SHA-1 engine, key generator and an RSA engine. The program code is the Core Root of Trust for Measurement (CRTM) and is used to measure various other platform components. Although ideally, it should be part of the TPM, it may be located in other firmware like the BIOS and is said to be part of the Trusted Building Blocks (TBB). The TPM also has an execution engine or some firmware to execute the program code including TPM initialization and measurement of various components. It also has some non-volatile storage which it uses to store keys, certificates and passwords.

![Image of TPM components]

Figure 2.1: The components of a TPM [9]

The TPM has a special set of registers called Platform Configuration Registers (PCRs) which are used to store information about the platform on which the TPM resides. The information about the platform state that is stored in a PCR is often called a ‘measurement’. Initially the PCRs 1-16 are initialized to zero and PCRs 17-23 are initialized to -1. Storing platform measurements in a PCR is termed as ‘extending a PCR’ and is done by concatenating the current value of the PCR with a digest of the new measurement and using the SHA-1 engine to create a digest of the result which is stored back in the same PCR. PCRs are reset on reboot. Finally, TPMs must be tamper resistant. According to the specification, it is sufficient to bind the TPM module physically to a platform such that it is difficult to disassemble or transfer it although it recommends tamper-evident mechanisms to be used in addition to these measures.

2.2.4 TPM keys

Key generation in the TPM framework has been clearly described in the TPM specification documents [10], [11], [12]. The Endorsement Key (EK) could be generated by the manufacturer or the TPM owner at first use. It is certified by the manufacturer to create the Endorsement credential. The Storage Root Key (SRK) is created when a user takes ownership of the TPM. Its usage authorization data is encrypted by the EK. The SRK is used to protect the storage of other TPM keys. An Attestation Identity Key (AIK) is generated in the TPM but is certified (or the Atestation Identity credential is created) by a Trusted Third Party (TTP) called a
Privacy CA (PCA) based on the Endorsement credential. The AIK in turn is used to certify other binding keys and signing keys. The constraints on h/w TPM key usage include:

a. The EK, SRK and AIK are non-migratable keys. They are bound to a specific TPM and their corresponding private parts are never available in plain text outside the TPM.

b. The private part of the EK and SRK never leaves the TPM. The EK and SRK can be used only for encryption, decryption and not for signing.

c. The private part of an AIK can be stored inside the TPM or outside the TPM protected by the SRK. It can only be used for signing data that originated from the TPM and not arbitrary data. This implies that it can only be used to certify non-migratable keys. It cannot be used for encryption or decryption.

Further details on other types of TPM keys are available in [9].

2.2.5 The TCG Software Stack (TSS)

Figure 2.2 shows the software stack defined by the TCG for accessing the TPM. The TDDLI provides an OS-independent interface for interaction with the TPM. It also facilitates inter-operation of different implementations of the TSS. There is only one instance of a TDDL per platform and it enforces single threaded access to the TPM. The TCSI provides a common interface to different TCG Service Providers that may co-exist on the same platform. The services it provides include context management, credential and key management and measurement event management. The TSP resides in the same address space as the application using the TPM. Authorization occurs at this layer. This layer also exposes a limited set of cryptographic functions using a C-interface.

2.2.6 TPM applications

TPMs are employed for two main purposes - the first is for secure storage and the second is for ensuring and proving platform integrity. Both the applications are described briefly below.

Secure storage

There are two mechanisms to achieve protected storage using a TPM. The first method called binding is essentially encryption of the data to be protected using the public key of an asymmetric key pair. If the private key of this key pair never leaves the TPM, it ensures that only the TPM that was originally used for encryption can be used to recover the original data. Binding using a migratable key has no special significance beyond encryption. The second method called sealing is an enhanced form of binding. Here, data is protected using encryption with a symmetric key. This symmetric key, is then concatenated with a select set of PCRs before encryption. This ensures that the symmetric key (and hence, the data) can be recovered only
Trusted computing

Figure 2.2: The TCG Software Stack [9]

when the PCRs are in the same state as they were when used for encryption. Thus, sealing not only binds the data to the TPM (like binding) but also binds it to a certain state of the platform.

Figure 2.3: TPM key storage [9]

The TPM has limited storage for keys and therefore, inactive keys can be moved out of the TPM in encrypted form. Management of the available key slots on the chip is done external to the TPM by an Key Cache Manager (KCM). These externally stored keys can be protected using the SRK or its descendant storage keys as
Platform Integrity Measurement

Platform integrity measurements are performed by an appropriately enabled kernel. A digest of each such measurement is extended into a PCR. The measurements are themselves stored (this is called a Stored Measurement Log (SML)) or regenerated at the time of integrity reporting. Any tampering of the SML can be detected and hence, it is not stored in the TPM. This also allows arbitrary number of measurements to accumulate in the SML because it is no longer constrained by the size of the protected storage in the TPM.

Initially, platform measurements were performed only at boot time as described in [13] and [14]. These used the 'measure and load' paradigm where critical components are first measured before they are loaded or executed, i.e., the BIOS (RTM) measures the boot loader, the boot loader measures the OS kernel, the OS kernel in turn measures any start-up application that it loads and so on. This establishes a chain of trust from the RTM to the last component that is loaded. Trusted GRUB from [http://sourceforge.net/projects/trustedgrub/] is an example of a boot loader that performs such measurements.

However, it is not sufficient to only make static boot time measurements because this would not detect changes to the state of the system after boot. Besides, if the boot process terminates on detecting any deviations from the reference values, updates and upgrades would be severely restricted. Several frameworks and mechanisms for dynamic measurements of changes to system state have been proposed.

Terra [15] provides a mechanism for dynamically incorporating changes to system state. It uses the notion of 'closed box VMs' and 'open box VMs' to distinguish between trusted VMs and ordinary VMs. A closed box VM’s storage is divided into attestable and non-attestable disks. Every component of a closed box VM that wants to be attested generates an asymmetric key pair. The component’s key pair it passed to its parent application, which signs a certificate containing the hash of the attestable parts of the component and its data. Such a certificate chain is established from the tamper resistant TPM upto the applications inside the VM. Verification of the chain of trust needs certificates of every component in the chain, the manufacturer’s certificate (or any related certificate) of the TPM and the vendor certificates for every application in the chain to verify the hash in that application’s certificate. This allows management of different versions of applications transparently.

IBM’s Integrity Measurement Architecture (IMA) [16] provides mechanisms to measure dynamic system properties by instrumenting OS kernel code appropriately with hooks that perform measurements. The authors discuss the insertion of measurement points for different types of applications (static executables, dynamically loaded libraries and kernel modules). The measurements are made after the appli-
cation is loaded into memory thus preventing ‘bait and switch’ attacks. Techniques for re-measurement (e.g., dirty tagging), measurement validation and performance has also been discussed in detail in this work.

In [17], a dynamic property collector extracts the structural and data constraints using the binary executable of an application. The dynamic measurement module checks for the validity of these properties at application load time. IMA is then used to protect the loaded application from malicious tampering.

Platform Integrity Reporting

The RTR not only manages storage of integrity measurements but also attests to the authenticity of the stored values based on the TPM’s identity. This is achieved by the creation of an AIK that is used to sign the PCRs. Establishing the authenticity of the AIK proves that authenticity of the TPM and hence, confirms that no integrity measurement (in the PCRs) have been tampered. The process of proving the authenticity of the AIK is called Attestation.

Remote attestation

In the research community, the term ‘remote attestation’ encompasses more than just proving the authenticity of the AIK to a remote party. Remote attestation is the process of proving system integrity to a remote host (verifier) using a TPM. The host under scrutiny returns its SML and its PCR values signed using its TPM’s AIK. This allows the verifier to check if the configuration of the host conforms to some security policy and that the PCR values match the configuration (which prevents the host from lying about its configuration). Approaches to remote attestation include verification of binary hashes of system configuration and software [16], property based attestation that uses property certificates from a TTP [18] and software-based remote attestation [19] that is used to verify structural and dynamic properties of programs by exploiting properties of programming languages. Several remote attestation protocols have been proposed [9], [16]. The protocol in [9] is vulnerable to old-configuration replay attacks because it does not include any freshness. This problem has been fixed in the protocol in [16] by including a nonce that is signed along with the PCRs. But this protocol is still vulnerable to relay attacks. Direct anonymous Attestation is a remote attestation mechanism that does not depend on a Privacy CA but still ensures that a TPM’s transactions cannot be linked [20].

2.2.7 Relay and Cuckoo attacks

The relay attack (Figure 2.4(b)) outlined in [21] highlights the challenges in ensuring that an attester does not masquerade the properties of some secure host as the properties of the host being attested. The relay attack is a generalization of a cuckoo attack (Figure 2.4(a)) [22]. The cuckoo attack deals with ensuring trust establishment in a local TPM, i.e., preventing an attacker from masquerading a remote TPM as the local TPM. The difference between the two scenarios is, unlike in the cuckoo attack where the verifier has physical access to the host being attested, in the relay
attack, the verifier only has remote access to the attested host. Solutions to the

cuckoo attack are easier because of this difference which allows the establishment
of physical secure channels. The proposed solution to prevent the relay attack is
to include a public identifier (public key certificate) while signing the PCRs [21].
In [23], the authors point out that even an attacker could also use any public key
certificate with the PCRs to impersonate the desired host.

(a) The cuckoo attack  (b) The relay attack

Figure 2.4: Cuckoo and relay attacks [22], [21]

The secure channel establishment solution in [23] itself assumes that an SSL
certificate is bound to a single physical host. The authors suggest using an SSL
certificate with a Subject Key Attestation Evidence (SKAE) [24] extension. But,
SKAE certificates may not be useful in a context where an SSL certificate is shared
by several physical hosts. For example, cloud providers may use a load balancer
allowing several physical hosts to share a single SSL certificate.

2.2.8 Issues with trusted computing

One of the main reasons for the limited popularity of trusted computing is the lack
of end-user privacy protection. Use of TPMs with unique identities (EK) allows
tracking of TPM transactions and hence, end-user activity. This has been mitigated
to an extent by the use of Privacy CAs but the realization of trusted CAs in practice
is difficult. Lately, the development of direct anonymous attestation has solved this
problem of dependence on Privacy CAs. Another issue with trusted computing is
that it may result in monopoly by certain vendors leading to high switching costs
and lock-in [25].

2.3 TPM virtualization

The application of trusted computing to virtualization poses an interesting set of
challenges. Traditionally, trusted computing components (hardware, software, pro-
tocols) have been designed for a single host. TPMs have limited resources and
software stacks have been designed to prevent simultaneous TPM access by multiple
entities. However, in virtualized systems, several VMs run on the same host (hard-
ware). An extension of TPMs for use with VMs is called TPM virtualization and
results in a virtual TPM (vTPM) design. A new technique for multiplexing TPM
TPM virtualization

access between VMs and a solution to the larger storage required to save the state of individual VMs are among the first challenges. It is also necessary to guarantee isolation between each VM’s TPM storage and protect it against unauthorized access. The scalability of using a single TPM with several VMs needs investigation. Finally, the semantics of existing trusted computing processes (like attestation, key creation and certification) and virtualization processes (like VM migration) require re-examination in the context of trusted computing on virtualized platforms.

Several designs for vTPMs and vTPM architectures have been proposed in the recent years. We discuss four such solutions as described in [26], [27], [28] and [29]. A comprehensive summary of various vTPM design solutions can be found here.

2.3.1 Software virtualization of TPMs: by Berger S, et al., 2006

In this work, the authors identify the requirements for a vTPM and propose a vTPM design that is compatible with the running vTPMs in memory or on a cryptoprocessor. This architecture has been implemented on the Xen hypervisor. Central to this architecture is a privileged VM (Dom0 in the case of Xen) dedicated to running vTPMs. This VM has access to the h/w TPM and co-ordinates all requests to it. This VM also runs a vTPM manager that manages all the communication between a VM and its vTPM. VMs can optionally be configured to use vTPMs. On starting a VM that is configured to have a vTPM, a corresponding vTPM instance is started as a user-space process in the privileged VM.

Figure 2.5: Software TPM virtualization architecture

Figure 2.5 shows the vTPM architecture proposed by the authors. Each vTPM instance is assigned a unique 4-byte identifier that never leaves the privileged VM. This unique number is mapped to a unique interrupt (number) that is assigned to the VM. The VM uses this interrupt to communicate with its vTPM. The vTPM id-interrupt mapping is stored in the XenStore in the case of the Xen hypervisor along with the VM to vTPM instance mapping. On receiving a vTPM request, the
backend driver prepends the instance number using the mapping table to the request. The communication is managed using a split device driver model. The front end driver resides inside the VM and the back end driver in the privileged VM. It uses a special feature in Xen called the ‘xen-bus’. The xen-bus allows the VM to map a portion of its memory as shared and allow the privileged VM to access it. _Since communication happens using shared memory, unauthorized access to vTPMs by VMs is not possible._

A root vTPM instance is used to spawn all new vTPM instances. According to the authors, it was designed to manage migration of asymmetric keys between vTPM instances and encryption of the vTPM state itself during migration. The vTPM state is protected by encrypting it with a symmetric key which is in turn bound to the hardware TPM. The integrity of the state is not protected on the disk. Each vTPM instance has its own set of PCRs. The PCRs of the hardware TPM which are common to all VMs is used with each vTPM’s PCRs during attestation. Currently, work to integrate the TPM Emulator (Section 2.3.6) into this framework is in progress. The key generation and hierarchy is not clear in the implementation in Xen 3.1.0 [30]. The emulator supports key generation but this functionality does not seem to be harnessed yet. In this work, the authors propose three solutions for vTPM key generation, namely,

a. Signing a vTPM’s EK (vEK) with h/w TPM’s AIK: The vTPM’s AIKs (vAIKs) are obtained as usual from a Privacy CA.

b. Signing a vAIK with h/w TPM’s AIK: No details about the generation of the vEK or the vTPM’s SRK (vSRK) have been discussed.

c. Generation of EK credentials for the vTPM by a TTP as in the case of a h/w TPM.

2.3.2 Hardware virtualization of TPMS: by Stumpf F, et al., 2008

Since software TPMS do not provide the same security guarantees as hardware TPMS, a vTPM design that leverages on the Intel TXT has been proposed in [27]. The new Intel VT-X/I processor has two new CPU modes, namely, VMX root in which the VMM runs and VMX non-root in which the guest VMs run. The VM Control Structure (VMCS) is loaded into the processor everytime a control transfer or transition between the VM and VMM occurs. The authors propose a multi-context TPM which is multiplexed between the VMs by the VMM. Each VM is assigned its own TPM context (called a TPM Control Structure (TPMCS)). The TPMCS structure is shown in Figure 2.6(a). The PCRs (0-15) corresponding to the VMM can be read by all VMs but written only in TPM’s privileged mode. These PCRs correspond to the host machine (boot procedure and VMM) which is common to all the VMs and are stored in a special part of the TPM.

Every time a VM issues a TPM command, the VMM loads the corresponding context identified by a unique label into the TPM. This allows direct native execution of TPM instructions. The VMM uses an interval timer to limit the duration
TPM virtualization

for which a VM can use the CPU and hence, the TPM. The VM-TPMCS mapping is managed by a back-end TPM driver in the VMM. Figure 2.6(b) shows the structure of the proposed enhanced h/w TPM. The storage of these TPMCS structures is protected by sealing it to the SRK which is stored in the root-structure and is accessible only in VMX root mode. Here, the vEK is assumed to be certified by the h/w TPM’s EK and is re-certified after migration.

Further, the TPM is also given two privilege levels - the lower level in which guest VMs operate only on their own TPMCS and the higher level which is used by the VMM. Transitions between the two levels can be managed either using a new 1-bit control register or using the Intel-VX CPU transitions. In order to prevent the VMM from sniffing off each VM’s owner password, all communication between the VM and the TPM is encrypted using a session key derived from the owner secret. The transitions between different modes of the TPM must be synchronized with that of the Intel-VX CPU. Since mode-transitions in the CPU do not directly reflect in the TPM, the VMM explicitly manages these transitions while scheduling the TPM for use by VMs. Also, since sensitive instructions executed in the TPM cannot trap into the VMM (because it does not have its own program counter), exceptions are used to ensure that privilege levels are respected.

2.3.3 Para-virtualized TPM sharing: by England P, et al., 2008

While the above approaches to TPM virtualization require separate resources (e.g., vPCRs, delegation tables, counters) for every vTPM instance, another approach would be to share a single h/w TPM ‘safely’ between several VMs. This option is explored in [28]. It requires modifying the TPM v1.2 interface. It overcomes the drawback of the software vTPM’s operations not being hardware protected and the need for a virtualization capable hardware TPM. In order to achieve TPM para-virtualization, some parts of the TPM need to be replicated (e.g., PCRs, counters)/partitioned (e.g., delegation tables for delegation of rights, non-volatile storage) for each vTPM while the others can be shared safely (e.g., EK, SRK, random number...
generator)/ multiplexed (TPM key slots) between them.

Figure 2.7: Para-virtualized TPM sharing

The para-virtualization module is shown in Figure 2.7. It can be located in the TCB (e.g., hypervisor). It uses the hypercall interface (used by applications inside a VM to access the VMM) to expose the TPM functionality to each VM. The module contains a scheduler that controls TPM access and a command filter to prevent unauthorized access to TPM functionality like resetting vPCRs and extending PCRs corresponding to the host VMM. It also includes a context manager that maintains VM-vTPM associations, isolates the TPM contexts of different VMs and manages resource handles. Furthermore, the module uses the VMM’s TPM driver to access the h/w TPM. The VMs use the corresponding HyperTPM driver to access the TPM. The para-virtualization module provides each VM with a set of vPCRs that are loaded into resettable h/w PCRs (available in TPM v1.2) for use. While all PCRs of the hardware TPM can be read by all VMs, a VM can modify or extend only its set of PCRs. The protection and location of the vTPM persistent storage is not clear. Finally, a new set of command structures that serve as a wrapper for existing raw TPM commands has been designed. The use of these modified commands requires changes to the TPM interface specification. The only implementation related details disclosed by the authors are the number of lines of code that was required to achieve para-virtualization. No details regarding the platform used for the implementation are available in the paper.

Finally, the vTPM key hierarchy design is unclear. The authors suggest sharing of the h/w TPM EK between all vTPMs instead of a separate vEK per VM. Similarly, the h/w TPM SRK can be shared between vTPMs. But it is unclear how AIKs and other keys are created and stored. An important advantage of this approach to
TPM virtualization is that it allows recursive TPM virtualization with each layer exposing a para-virtualized interface to the layer above it.

2.3.4 Property-based TPM virtualization: by Sadeghi A, et al., 2008

In [29], the authors only define a logical architecture for vTPMs without any implementation specifications. The previous vTPM designs use binary platform measurements, which allows migration only between identical hosts rather than hosts with comparable security properties. In order to overcome this limitation, the authors propose the use of property-based measurement and attestation. The design assumes that the underlying VMM protects the vTPM’s state and operations against unauthorized access. The vTPM (shown in Figure 2.8) consists of a number of ‘property providers’ which interpret and store information in a certain format (binary, using property certificates, etc.). Each provider has its own set of virtual PCRs which it extends with measurements. The property filter gives the flexibility of choosing the format and content of the PCRs that is exposed. The vTPM design also includes a key management module that generates keys itself or delegates it to the hardware TPM. Each vTPM itself generates its vEK and vSRK which are then certified by a local Certificate Authority (CA).

Figure 2.8: vTPM structure using parameter based TPM virtualization [29]

The PCR values of the host platform which is common to all VMs is mapped by the different property providers to their respective vPCRs. The authors also outline how data can be sealed using these property providers. Designs of protocols for property-based attestation can be found in [18] and [31]. Although it is mentioned that a VM accesses its vTPM via an interface, the design does not address the problem of associating VMs with their vTPMs. No specific security measures for the vTPM storage have been outlined because the design assumes, it is protected by the VMM. The location of this storage (in the VMM, filesystem, etc.) is not clear either.
2.3.5 TPM Virtualization: Building a General Framework: by Scar-lata V, et al., 2007

In this work, the authors define the components of a vTPM architecture. Figure 2.9 shows their vTPM architecture with generalized components. The framework uses the h/w TPM to ensure integrity of the VMM and other framework components. This trusted VMM is used to protect the execution of the vTPM and the ’protected persistent storage’ protects the storage of vTPMs. The vTPMs can be implemented either in the VMs or in dedicated hardware or in a combination. This involves a trade-off between performance (the in-VM implementation is better for performance) and security (the hardware implementation is better for security). The vTPM manager is responsible for communication between the TPM and individual vTPMs, ensuring the integrity of vTPM code and managing vTPM access to resources like the protected persistent storage.

Figure 2.9: Generalized vTPM architecture

The authors also discuss the details of the content of different credentials (e.g., endorsement credential, attestation identity credential). They propose that for static creation of vTPMs, the TTP like a CA can be used but the vTPM factory can generate credentials for dynamically created vTPMs. The vTPM factory creates a vEK for the vTPM certified using an AIK from its h/w TPM. The corresponding endorsement credential is generated by the CA after verifying the platform’s integrity using TPM PCRs. A verifier has to not only check the credentials of the vTPM, but also the credentials of the corresponding vTPM factory. Therefore, during
attestation, the verifier checks the vPCRs of the VM followed by the PCRs of the platform.

2.3.6 The TPM emulator

The TPM emulator [33] was originally designed to be used as a test tool that could be used instead of a h/w TPM for academic purposes. Its interface is compatible with that of TPM version 1.2. It also includes a new TPM device driver that allows interaction with a software TPM instead of a h/w TPM. It consists of a kernel module that is used to start a daemon that caters to TPM requests. The emulator uses a file on the file system for non-volatile storage. This file is read during every start-up in “save” mode but is written only when the emulator is shutdown. Any changes to the storage are cached in memory till the next shutdown. Due to the emulator’s original design goal as a test tool, its permanent storage file containing the EK, SRK and other persistent data is protected neither in memory nor on the filesystem.
<table>
<thead>
<tr>
<th></th>
<th>Hardware virtualization</th>
<th>Software virtualization</th>
<th>Para-virtualization</th>
</tr>
</thead>
<tbody>
<tr>
<td>vTPM NV store</td>
<td>Outside the TPM</td>
<td>Outside the TPM</td>
<td>Not clear</td>
</tr>
<tr>
<td>vTPM NV confidentiality</td>
<td>Protected by sealing to the h/w TPM state</td>
<td>Protected by sealing to h/w TPM</td>
<td>Not available</td>
</tr>
<tr>
<td>vTPM NV integrity</td>
<td>Using digital signatures</td>
<td>Not integrity protected</td>
<td>Not integrity protected</td>
</tr>
<tr>
<td>vTPM execution</td>
<td>In the h/w TPM</td>
<td>In the memory (of the VMM or VM)</td>
<td>In the h/w TPM</td>
</tr>
<tr>
<td>Hardware requirements</td>
<td>Virtualization enabled TPMs Special processors (e.g. Intel TXT)</td>
<td>Current h/w TPM</td>
<td>Current h/w TPM</td>
</tr>
<tr>
<td>Software requirements</td>
<td>Trusted VMM support</td>
<td>Trusted VMM support</td>
<td>Modification of TPM API Trusted VMM support and TPM driver modification</td>
</tr>
</tbody>
</table>

Table 2.1: Comparison of different approaches to TPM virtualization
Chapter 3

Overview and analysis of existing VM-vTPM migration protocols

3.1 Motivation for secure VM-vTPM migration

One of the most important benefits of virtualization is Virtual Machine (VM) migration. VM migration facilitates user mobility, load balancing, managing failures and system management in general. VM migration is now supported by several hypervisors like VMware’s vSphere Hypervisor (ESXi) and the Xen hypervisor. Research in the area of VM migration mainly focused on optimizing migration performance through live migration as reported in [34], [35] and [36]. While the semantics and performance of live VM migration are well explored, the security aspects have received very little attention.

The extension of trusted computing to virtualized systems using vTPMs allows applications in the VM to use the vTPM for secure storage and reporting platform integrity. In order to ensure their correct operation after migration, the vTPM must be migrated along with the VM. We refer to this as Secure VM-vTPM migration. The proposals for vTPM designs have been accompanied by proposals for vTPM migration. In this chapter, we first briefly describe Xen’s support for VM migration. Then, we discuss the proposed vTPM migration protocols and derive a set of security requirements from them.

3.2 Xen’s existing VM migration protocol

The Xen hypervisor allows migration of live or suspended VMs. Here, we discuss only Xen’s migration of suspended VMs. Suspended VMs can be migrated using an unencrypted or an SSL connection. The existing implementation uses Python and C. The migration protocol using SSL uses pyopenssl (a Python wrapper over OpenSSL). Currently, it does not enforce authentication by any means (e.g., certificate verification, login). The Xen implementation assumes that the filesystem of
the migrating VM is located on shared storage that is accessible to both the source and the destination of the migration. Therefore, it transfers only the RAM image of the VM. The source saves the memory image of the migrating VM directly over the network on the destination which restores the VM.

3.3 Review of the existing vTPM migration protocols

Recent work on vTPM migration includes protocol proposals in [26], [27] and [29] and a protocol implementation in the Xen Hypervisor [30]. Here, we briefly review these protocols and analyze their security properties.

3.3.1 Protocol 1: by Berger S, et al., 2006

![Figure 3.1: The vTPM migration protocol by Berger S, et al.](image)

In [26], the authors assume that the destination is trustworthy. The protocol as shown in Figure 3.1 has been proposed for migration between identical platforms. The authors state that it can be used alongside live VM migration. A migration controlling process initiates the transfer by creating a new vTPM instance at the destination. Then, it creates a nonce and transfers it to the source in encrypted form. The key used for encryption is not clear. At the source, this nonce is used to lock the vTPM to prevent further changes to it. The vTPM is then encrypted using a newly generated symmetric key, which is in turn encrypted using the vSRK of the vTPM’s parent instance. The encrypted state information includes keys, counters, any permanent flags, authorization and transport sessions and data. A hash of each of the above parts is added to an internal migration digest. The vTPM is deleted from the source and the encrypted state is transferred to the destination with the migration digest. The authors state that the vSRK of the parent vTPM instance
Review of the existing vTPM migration protocols

is transferred to the destination using mechanisms applicable to migratable TPM storage keys [24]. At the destination, the received binary object is decrypted to extract the vTPM state. The digest is verified and if no violations are detected, the vTPM is unlocked using the nonce and restarted. Since the vTPM keys are assumed to be independent from the h/w TPM keys, no key regeneration occurs.

3.3.2 Protocol 2: By Stumpf F, et al., 2008

Figure 3.2: The vTPM migration protocol by Stumpf F, et al. [27]

Figure 3.2 shows the migration protocol proposed in [27]. Here, the authors assume that a trusted communication channel exists between the source and the destination. The entire migration is assumed to occur between identical platforms inside this trusted channel. The source and the destination communicate through their respective migration interfaces. The source creates a non-migratable TPM key $K_A$ that is certified using an AIK. It sends $K_A$'s certificate along with a nonce ($N_A$) to the destination. The destination verifies the certificates for $K_A$ and its corresponding AIK. It creates a non-migratable TPM key $K_B$ and certifies it using an AIK. Then, it transfers the certificates along with a fresh nonce ($N_B$) and $N_A$. The nonces are encrypted using $K_A$. The source verifies certificates for $K_B$ and its corresponding AIK. It then encrypts the vTPM to be migrated along with both the nonces using $K_B$ and transfers this to the destination which decrypts it to retrieve the vTPM and verify the nonces. Here, the authors recommend re-signing only the vEK at the destination and retaining all the other keys and certificates.

3.3.3 Protocol 3: inferred from the source code of the Xen hypervisor

In [30], the migration protocol is very similar to the protocol proposed in [27]. Figure 3.3 shows the migration protocol as implemented in Xen. The migration manager at the source initiates the transfer by requesting the migration key of the destination. The migration key of the destination in this case is a global asymmetric key which is used to protect the storage of all instances of vTPMs in non-volatile memory at the destination. It retrieves the vTPM to be migrated, encrypts it using the migration key obtained and transfers it to the destination. The destination
decrypts the obtained blob to retrieve the vTPM and stores it. Since the vTPM key hierarchy is itself not clear, key regeneration after migration is not addressed.

### 3.3.4 Protocol 4: by Sadeghi A, et al., 2008

The migration protocol proposed in [29] is shown in Figure 3.4. In this protocol, the vTPMs are created using a property based virtualization mechanism which uses property certificates rather than binary hashes as measurements for extending vPCRs. This allows migration of vTPMs between platforms with the same security guarantees instead of between identical platforms (as in the other protocols). Two migration controlling processes, one each at the source and the destination,
co-ordinate the migration. The source initiates the migration by creating a new vTPM on the destination. It establishes a trusted channel with the destination and obtains a key $K_{bind}$ that is bound to the configuration of the destination platform. It is not clear how the source verifies the binding of $K_{bind}$ to its platform. It creates a new symmetric key $sk$ and encrypts the state of the vTPM with it. It then encrypts $sk$ with $K_{bind}$, and transfers both the encrypted blobs to the destination. The destination first retrieves $sk$ and uses it to obtain the vTPM state. The key $sk$ and the vTPM are deleted from the source before it transfers them in encrypted form. No key regeneration occurs here because the vTPM keys are independent of the h/w TPM (platform).

3.4 Evaluation of the proposed protocols

We evaluate the solutions in Section 3.3 with respect to the context in which they can be used, the security of the actual vTPM transfer (confidentiality, integrity and replay resistance), possibility of generation of duplicate copies or loss of data and finally, their privacy implications.

3.4.1 The lack of a context

The authors of the above discussed protocols do not specify the context (setting) in which their migration protocols can be used. This context includes the entities involved in the migration, their trust relationships and the adversary’s capabilities. Specification of this context is essential for evaluating the practicality and security of these protocols.

VM migration use cases

Broadly, one can imagine two scenarios for VM migration (and hence, vTPM migration).

(a) Inter-cloud migration

(b) Intra-cloud migration

Figure 3.5: User authorized inter-cloud migration and provider initiated intra-cloud migration

23
a. Inter-cloud migration: An example of inter-cloud migration has been depicted in Figure 3.5(a). This refers to user authorized direct migration between two hosts that are owned by different cloud providers A and B. Direct transfers between the two providers allows tracking changes to customer base. Anonymous transfers have legal implications. Another important aspect is the trust between the providers. A malicious provider could cheat by relaying the properties of a secure machine as described in the relay attack (Section 2.2.7). This is very difficult to detect without physical access to the provider’s cloud. Therefore, such direct transfers between clouds administered by different cloud providers are unlikely.

b. Intra-cloud migration: Figure 3.5(b) shows an example intra-cloud migration initiated by a provider. This refers to the case where the source and destination providers are the same. It may be transparent to the end user. Such migrations may occur during load balancing, security policy changes, etc. This is a more likely application of VM migration.

The attacker model

None of the protocol designs are accompanied by a description of the attacker’s capabilities which is necessary for their security evaluation. Here, we assume a realistic attacker model where an attacker not only has access to network data but can also modify or inject messages into the network. We also assume that he is computationally bounded and hence, brute force attacks on cryptographic schemes are difficult. We also assume that the attacker does not have physical access to platforms between which the migration occurs.

Since intra-cloud migration is a more realistic use case than inter-cloud migration, further evaluation of the above discussed protocols here assumes an intra-cloud setting. The attacker model consists of a computationally bounded adversary who can obtain all network data, modify and inject messages but does not have physical access to the platforms between which the migration occurs.

3.4.2 Secure VM-vTPM association

All the solutions to vTPM migration in Section 3.3 lack details regarding the binding of a vTPM to its VM during/after the migration, the semantics of VM migration (whether the VM is migrated in its live state, suspended state or in a powered off state) and the timing relationship between the vTPM and VM migration and resumption at the destination. The vTPMs and VMs also have unique identifiers which could lead to conflicts at the destination during migration.

Binding a vTPM to its VM is important to resume operation at the destination and to prevent an attacker from modifying the mapping of vTPMs to their VMs when several VMs and vTPMs are transferred in one session. The binding could be implicit if a single VM-vTPM pair is transferred in one session. Else, explicit mechanisms such as including the VM’s name in the vTPM will be required as binding
Evaluation of the proposed protocols

information. Thus, the relative timing of vTPM and VM migration influences the type of VM-vTPM binding required.

The semantics of VM migration are important because any changes to the VM should be synchronized with the vTPM. For the migration of a powered off VM or a suspended VM, only a secure transfer protocol is required but live migration is more complex. This is due to the problem of synchronizing VM changes with the vTPM which makes the relative timing of vTPM and VM resumption at the destination important. Since some live migration techniques allow the VM to be started on the destination before it is stopped at the source [34], ensuring consistency between VM state and its vTPM is very difficult. Also, secure migration may not start the VM at the destination immediately after the transfer. This complicates the usage of live VM migration with vTPMs. Extending live VM migration with vTPM migration does not seem as straightforward as the authors claim it is, in [20].

It is important to lock the suspended VM and its vTPM before their transfer, to ensure no changes occur in either of them during the transfer. The protocol in [26] incorporates such a locking mechanism before the transfer. The implementation in [30] locks the vTPM before any operation on it. But the protocols in [27] and [29] do not include any mechanisms to prevent changes to the vTPM during migration.

All the existing vTPM designs and implementations make use of a unique identifier to identify a vTPM instance. During vTPM migration, if the id of the migrating vTPM matches the id of any existing vTPM on the destination, resolution procedures to handle such a conflict are necessary. In the protocols in [26], [27] and [29], it is not clear how an id-clash is handled. An easy but inefficient solution would be to reject the incoming vTPM. This is the approach used in [30]. A more effective solution would be to re-assign a unique identifier at the destination. If the unique identifier is a number and is assigned sequentially (incremented by one) as in [30], exposing it could lead to load estimation attacks.

In summary, live migration of VMs that use vTPMs is difficult. Suspended VM migration is simpler but not trivial. Also, transferring the VM along with the vTPM simplifies their re-association at the destination. It is also important to lock the vTPM and the VM to prevent any changes to them during migration. VM/vTPM identifiers should not be exposed and if exposed should not leak additional information by appropriate design (E.g. randomizing the vTPM ids to prevent load estimation).

For the rest of this section, we examine the security properties of the protocols outlined in the Section 3.3 under their application to intra-cloud, suspended VM migration with respect to the adversary model in Section 3.4.1
3.4.3 Authenticated transfer to a secure destination platform

While protocols in [26] and [30] lack destination and source authentication before vTPM transfer, [27] and [29] establish secure channels with the destination but the subsequent use of this channel for the vTPM transfer is unclear. Also, in [29], a new vTPM instance is created at the destination even before a trusted channel is established. Lack of destination authentication allows Man-In-The-Middle (MITM) attacks. Not using an established channel for secure sessions can lead to ‘bait and switch’ attacks. In either case, an attacker can get access to a plain text vTPM from which he may be able to derive the VM’s configuration information (like the details about the operating system used by the VM). He may also obtain all the vTPM keys which could be re-used to impersonate a secure execution environment. If the attacker also gets access to a plain text VM along with a vTPM, he may get access to any confidential data stored on the VM (personal information, passwords in browsers, etc.). He could impersonate any publicly accessed service that was hosted by the VM and impersonate the VM itself if he obtains its X.509 certificate during the transfer. Furthermore, lack of authentication could lead to data loss if an attacker impersonates the intended destination but does not replay the vTPM to it. Missing source authentication allows repudiation of the transfer by the source which may have legal consequences.

In order to ensure that the VM and its vTPM are migrated only to a secure destination, it is necessary to verify its integrity using remote attestation. But some of the existing proposals for vTPM migration either assume the trustworthiness of the destination ([26]) or completely ignore the relevance of verifying the security properties of the destination ([30]). In [27], the authors assume the verification to be part of the establishment of the trusted channel like in [23] while in [29], it is not clear whether attestation is part of the secure channel establishment.

During migration, in order to prevent MITM attacks that can cause VM and/or vTPM loss or leakage of VM and/or vTPM information, it is important to establish an authenticated channel between the source and the destination and bind the transfer of the VM and its vTPM to it. Furthermore, the source must use remote attestation to verify the integrity of the destination platform. Protocols for remote attestation have been discussed in Section 2.2.6

3.4.4 Confidentiality and integrity preserving and replay resistant transfer

All the protocols in Section 3.3 ensure the confidentiality of the vTPM’s content. While [27], [30] and [29] encrypt the vTPM state using a non-migratable key tied to the destination’s h/w TPM, the protocol in [26] uses a symmetric key for encryption and this symmetric key is itself encrypted using a migratable source TPM key.

The protocols in [27], [29] and [30] do not ensure the integrity of the transferred vTPM like the protocol in [26], which protects integrity by creating and transferring a digest of each part of the vTPM. Lack of integrity verification can lead to modify
Evaluation of the proposed protocols

and replay attacks. An attacker may modify only the vTPM. For instance, changing just the vTPM of a VM hosting a public service such that attestation fails can lead to Denial of Service (DoS) attacks. It may also be possible to modify both the VM and its corresponding vTPM such that they are consistent with each other before replaying it to the destination. For example, an attacker may modify a VM to act as an email proxy and use it for creating spam emails for which the original owner may be held liable.

The protocol solution in [27] prevents replay and overwrite attacks using nonces to ensure freshness. In [26], the vTPM is locked using a nonce generated at the destination to prevent replays. The other solutions in Section 3.3 do not include any mechanisms to prevent replays. If an attacker can replay an old version of a VM and its vTPM, it could cause data loss. An attacker can replay a modified version of the VM and its vTPM (and overwrite the original copies) if source authentication and integrity checks are missing. For example, the attacker could modify the VM to allow him to monitor activity on it or steal data.

In summary, it is important to protect the confidentiality and the integrity of the VM and its vTPM during migration using appropriate cryptographic techniques like encryption and digital signatures respectively to prevent unauthorized access and modification. Replay attacks should also be prevented using nonces.

3.4.5 Atomicity of the migration

‘Atomicity’ refers to the property of ensuring successful migration or maintaining status quo without allowing any intermediate state. It includes preventing generation of duplicate copies of vTPMs and loss of data. The protocols in [27] and [30] do not delete the migrated vTPM from the source. This results in an increased number of copies of the vTPM after each migration. In [26], the protocol deletes the vTPM state information completely before transfer but does not specify any recovery mechanisms in case the migration fails. In [29], the authors suggest storing the encrypted state persistently till the transfer is successful. But storing only the encrypted vTPM state and not the key required to decrypt it, also makes recovery impossible. Lack of recovery mechanisms can result in DoS to the VM/vTPM owner. None of the protocols in Section 3.3 address deletion of the VM and its vTPM from the destination, only if the migration is not successful.

Ensuring deletion at the source in case of successful transfer and deletion at the destination in case of failure are required to ensure atomicity of the migration. It is critical for failure recovery and to prevent duplicate copy generation. It is non-trivial to ensure deletion and in intra-cloud scenarios, it is assumed that the source and the destination are trusted to perform deletion of the vTPM when required.
3.4.6 vTPM key hierarchies: Validity and implications for migration

vTPM key hierarchies are important for vTPM key migration. Here, we examine the validity of the proposed hierarchies proposed along with different vTPM designs (Section 2.3), the generation of vTPM credentials (which are essentially public certificates) and their content and finally, how key hierarchies affect migration.

Validity of existing vTPM key types and hierarchies

None of the vTPM designs (Section 2.3) elucidate the types of keys that could be used for the vEK, vAIK and vSRK. Due to the constraints on TPM key usage (Section 2.2.4), vTPM key types have to be different from those used by TPMs. According to the authors of [32], the vEK and the vSRK have to be legacy keys (which are keys generated and used outside the TPM for both encryption and signing) and the vAIK has to be a signing key. But such a generalization is not possible because it depends on where the keys are generated and who signs them. For instance, if a vEK is signed using a h/w TPM AIK, it becomes a non-migratable key and not a legacy key.

The key hierarchy in [27], where the h/w TPM’s EK is used for signing the vEK violates the TPM specification which states that the h/w TPM EK should not be used for signing. All the options in [26] are valid. In [29], the authors propose separating the vTPM key hierarchy from the h/w TPM hierarchy which is valid.

Generation of credentials (certificates) for vTPM keys

Although making the vTPM and the TPM key hierarchies disjoint seems elegant and efficient at first glance, the basis for issuing such a certificate is difficult to define such that it preserves the independence of the h/w TPM and vTPM. However, when the vTPM keys depend upon the h/w TPM, the h/w TPM and the vTPM’s environment can be used as the basis to issue a certificate to the vEK/ vAIK. It is also hard to find a trusted party to issue such certificates that are valid across platforms when the hierarchies are disjoint but this problem can be solved in an intra-cloud usage scenario. If the vTPM key hierarchy depends on the h/w TPM key hierarchy, the platform itself could issue these certificates. The exact content of these certificates and their validity period is not easy to define meaningfully when the hierarchies are not linked. However, when they are related, it is sufficient to have the vTPM key, signature on the vTPM key by the issuer (platform), the entity to which the certificate is issued (discussed below) and the validity (valid as long as the vTPM resides on that platform) on the certificate. Finally, another disadvantage of retaining the same keys across migrations is that it allows the source to track the vTPM/VM if the keys are reused.

In all of the above protocols, the entity to which the vTPM key (vTPM or VM or vTPM-VM or platform) is issued is unclear. vTPMs can be copied easily and hence, issuing keys to them is unacceptable. Issuing vTPM keys to VMs reduces
Evaluation of the proposed protocols

the flexibility of vTPM usage and could potentially aid tracking, which is also the case if the keys are issued to the vTPM-VM pair. Although vTPM certificates can be issued to the platform that hosts them, it would allow tracking of vTPMs to a platform. Therefore, it is best to include only the key type in the subject field of a vTPM key certificate and leave it to the verifier to check the issuer’s properties.

vTPM key hierarchies and migration

The options in [26] are re-examined in view of vTPM migration.

a. Signing a vEK with h/w TPM’s AIK: This makes the vEK non-migratable. It will have to be regenerated at the source or the destination. The Privacy CA could then verify the new vEK and allow retention of the old vAIKs or the vAIKs could be regenerated. Binding the vEK to the h/w TPM AIK and obtaining vAIKs from the Privacy CA increases dependence on the Privacy CA. Also, ensuring that the Privacy CA is informed every time the vEK changes (so that it can revoke the corresponding vAIKs or allow their reuse with the new vEK) is difficult.

b. Signing a vAIK with h/w TPM’s AIK: It will require regeneration of all vAIKs. The keys signed by the vAIKs can either be recertified or regenerated at the source or the destination. Due to reduced dependence on the Privacy CA, signing the vAIK with the h/w TPM AIK is a better solution, although it increases the number of requests to the h/w TPM.

c. Generation of EK credentials for the vTPM by a TTP as in the case of a h/w TPM: This obviates the need for key regeneration after migration provided the basis for issuance of such credentials is independent of the platform on which the vTPM resides.

Key regeneration is non-trivial, especially for suspended VMs. Processes that are using vTPM keys on the source will have to be updated to use new keys. This might mean re-encrypting data and re-encoding handles into applications that have been frozen as part of the VM suspension, modifying persistent data of applications that reuse keys protected by the vTPM, etc. The Trusted Computing Base (TCB) will have to be aware of all applications and their key associations. Besides, signing a vTPM key with the h/w TPM AIK would require creating and storing a h/w TPM AIK at least for every vTPM (to prevent tracking of vTPMs to the same h/w TPM). The number of keys that can be generated is limited by the amount of space available in the h/w TPM.

A vTPM key hierarchy that is independent of the corresponding h/w TPM hierarchy is difficult to define such that it complies with existing TPM key hierarchy semantics. vTPM key types depend directly upon the definition of the hierarchy. The proposed vTPM key hierarchies that depend on the h/w TPM force key regeneration which is non-trivial. Signing a vAIK with a h/w TPM AIK is better than signing a vEK with a h/w TPM AIK despite its performance inefficiency because it reduces dependence on the Privacy CA.
vTPM key hierarchies and performance of migration

Since a single h/w TPM is multiplexed between several vTPMs (or VMs), frequent requests to the h/w TPM may cause performance problems. Although migration requests may be infrequent, alongside attestation requests and PCR extension requests, the h/w TPM could experience considerable load. It would be desirable to minimize the involvement of the h/w TPM in the vTPM creation, usage and migration process. Signing a vAIK using a h/w TPM AIK could create considerable load on the h/w TPM. Signing a vEK with a h/w TPM AIK is slightly better since it happens only once for each vTPM but increases dependence on a Privacy CA for generating vAIKs. Keeping vTPM keys independent of the h/w TPM is best for scalability but is not a good choice (Section 3.4.6). Thus, defining a suitable vTPM key hierarchy and a vTPM migration protocol will involve trading off some security guarantees for performance.

All migration solutions in Section 3.3 involve the h/w TPM in the retrieval of the vTPM at the source and its storage at the destination. This is because all of them seal or bind the state of the vTPM to the h/w TPM for secure storage. This makes it imperative to involve the h/w TPM while unsealing it at the source and sealing it at the destination. All the implementations also use a h/w TPM non-migratable key to encrypt the vTPM during migration. While this provides better security, using a key not bound to the h/w TPM will not undermine security if the destination and the source platforms have mutually attested each other in a prior step.

Signing the vEK with the h/w TPM AIK is better than signing each vAIK with the h/w TPM AIK for performance but increases dependence on a Privacy CA. A tradeoff between security, performance and dependence on third parties like a Privacy CA is unavoidable in the design of the vTPM key hierarchy. Although non-migratable TPM keys are more secure than keys that can be used outside the TPM, their frequent use may lead to performance problems by overloading the h/w TPM. Instead, it is reasonable (from a security perspective), to delegate trust to an environment that can generate and use keys without involving the h/w TPM. Therefore, non-migratable TPM keys must be used only when absolutely necessary and the delegation of trust must be used for better performance.

3.4.7 Privacy preserving migration

Despite its various advantages, the adoption of TPM technology was not quick largely due to privacy issues. Therefore, it is important to ensure that the vTPM migration protocol does not undermine privacy or cause unnecessary data leakage.

(v)TPM Tracking

Currently, authenticated communication over the network uses a fixed, public identifier issued by a TTP (like a X.509 certificate). H/w TPM AIKs used alongside such a public identifier will always be linkable to the identifier. Whether they are linkable to a certain h/w TPM, depends upon how many physical TPMs are linked
to that identifier. If there is just one h/w TPM associated with the public identifier (E.g. one-to-one host-SSL certificate mapping), then all its transactions can be linked. On the other hand, reusing a key that is unique to a h/w TPM like the EK, even alongside different public identifiers also allows linking of transactions. Linking transactions to a single h/w TPM allows linking all transactions to a single platform. On the other hand, linking transactions of vTPMs allows trace back to the same VM. It also allows tracing back to the same platform if the vTPM is strongly bound to the h/w TPM.

Privacy implications of the use of the protocols in [26], [27], [29] and [30] are important for their adoption and use but have not been addressed in earlier work. None of the implementations regenerate vTPM keys across migrations. Retaining keys across migrations may allow the source to track vTPMs transactions. Another disadvantage of retaining such keys is that a source can recognize a VM (say, when it is migrated back to it) that it hosted earlier even if the state of the VM is not the same.

However, in intra-cloud migration scenarios, tracking by the source platform may not be as significant as tracking by a third party. Reuse of vTPM keys for communication (on the same host or across migration on different hosts) should be avoided. This implies that all keys that are used by applications for any communication like vAIKs for attestation, keys signed by vAIKs for public identity, session encryption or signature generation, etc. should not be reused. Keys that are used only internally within the VM and vTPM like storage keys can still be reused.

Information leakage

In the implementation in [30], the migration key whose public part is transferred from the destination is actually the asymmetric key used to protect the storage of all vTPM instances (global vTPM SRK). The same key is reused for every migration protocol instance. This could allow tracking of which vTPMs were on which host during migration. Such information can be used in co-location attacks [4]. Similar information leakage occurs if any of $K_A$, $AIK_A$, $K_B$ or $AIK_B$ are reused in the protocol in [27] or if $PK_{bind}$ is reused in the protocol in [29].

Other forms of information leakage should also be avoided. In [26], the parent instance which is used to securely store the vTPM state and keys is migrated to the destination. If many vTPMs have a common parent instance (like in [30]), this could allow an attacker with access to the vTPM infrastructure to retrieve the keys and the state of the sibling vTPMs.

The adoption of vTPMs could face considerable resistance if end-user privacy is not protected. Source tracking is not a major threat in intra-cloud migration. In order to prevent tracking by a third party, certain vTPM keys must not be reused and hence, not migrated with the vTPM. However, all vTPM keys do not have to be regenerated on every migration because it is safe to migrate some keys which are
used only within the VM or vTPM. Further, one must ensure that no unnecessary information is revealed during the migration.

Inter-cloud migration is similar to intra-cloud migration but more complex on several fronts. The source and the destination environments must be identical which is difficult to ensure across providers. Privacy requirements may be more stringent but source tracking is harder to prevent because ensuring VM and vTPM deletion is difficult without physical access.

### 3.5 Requirements for a migration protocol

The first step towards the design of a vTPM migration protocol is the formulation of the properties that characterize it. Explicit definition of the security requirements also provides a basis for the evaluation of a protocol after design. Here, we enumerate the security requirements for a migration protocol in an intra-cloud, suspended VM (and its vTPM) migration scenario.

a. **Independence from the integrity measurement mechanism**: A migration protocol must be generic and must not be tied to any specific integrity measurement mechanism (binary measurement or property-based measurement).

b. **Authenticated data transfer between secure platforms**: An attacker must not be able to launch MITM attacks or bait and switch attacks. Also, the attacker must not be able to migrate a vTPM from a secure platform to his insecure platform or vice-versa.

c. **Secure VM-VTPM association**: An attacker must not be able to modify the mapping of VMs to their vTPMs without the change being detected.

d. **Confidential data transfer**: An attacker should learn nothing about the contents of the communication between the source and intended destination except for their public identities and the existence of a conversation.

e. **Integrity preserving data transfer**: An attacker should not be able to modify the VM and/or its vTPM without the modification being detected.

f. **Replay resistance**: An attacker should not be able to replay an old communication sequence successfully without the replay being detected.

g. **Source non-repudiation**: It must not be possible for the source to deny the migration.

h. **Atomicity of the transfer**: The entire migration should be an atomic operation: either the entire migration process completes or status quo is maintained. No intermediate state (copy of the vTPM at both source and the destination or data loss) must result from the migration.
i. **Privacy preserving data transfer**: It must not be possible for an attacker to link two migrations from the same physical host unless the host has a fixed, unique network identity (public key certificate, fixed IP address, etc.). It must not be possible for a host to track the transactions of a vTPM after it has been migrated. Ideally, a host must not be able to recognize a VM and its vTPM that it previously hosted using vTPM keys but we relax this requirement for intra-cloud migration. Finally, transferring many VM-vTPM pairs in the same session allows tracing different VMs to the same physical platform. Therefore, only one VM-vTPM pair must be migrated per session.

Preventing VM and vTPM duplicate generation along with recovery mechanisms to prevent data loss, can be used to ensure atomicity of the migration. Also, appropriate resolution procedures will be required if VM and vTPM name or id conflicts occur at the destination. It is important in practice for the protocol to scale well with increasing number of migration and attestation requests.
Chapter 4

Migration Protocol Design

In this chapter, we present a novel vTPM key hierarchy which not only prevents vTPM transaction linking but also facilitates migration. Then, we propose a vTPM migration protocol based on the requirements outlined in Section 3.5 and our key hierarchy. A high level design of the protocol is discussed followed by descriptions of design alternatives and choices for each phase of the protocol. Finally, a security analysis of proposed protocol with respect to the requirements is presented.

4.1 vTPM key hierarchy design

The existing designs of vTPM key hierarchies either completely separate it from the h/w TPM key hierarchy or force key regeneration across migrations by linking the two hierarchies using the h/w TPM AIK. From Section 3.4.6, we know it is better to connect the two hierarchies. Section 3.4.7 indicates that certain vTPM keys may be retained without undermining privacy but the latter approach does not allow vTPM key reuse. To overcome these disadvantages, here, a new vTPM key set and hierarchy is defined. Then, the possible subset of these keys that may be contained in the vTPM of a suspended VM and the privacy impact of their migration is discussed.

4.1.1 vTPM keys

From the discussion in Section 3.4.6, we know that the vEK or the vAIK should be strongly bound to the h/w TPM AIK. The vSRK of each vTPM that is used to store its vAIKs and the other keys, can itself be protected using a non-migratable h/w TPM key, obviating the need for a vEK to protect it. Although signing the vEK with the h/w TPM AIK is better for performance than signing the vAIK with the h/w TPM AIK, it increases dependence on the Privacy CA for vAIK generation (Section 3.4.6). We bind the vAIK directly to the h/w TPM instead of the vEK to minimize dependence on the privacy CA. Hence, our proposed vTPM key hierarchy does not include a vEK.

The proposed vTPM key hierarchy is shown in Figure 4.1. All keys shown here are asymmetric keys. Here, a green line from key A to key B indicates key A is
Figure 4.1: vTPM key set and hierarchy

used to sign key B’s certificate (create a certificate (credential) for key B) and a red line from key C to key D indicates key C is used to encrypt the private part of key D.

Figure 4.1(a) shows the keys that are used only locally inside the VM. This includes the vSRK of a vTPM which is protected by a global binding key (gSRK) common to all vTPM instances. The vSRK can alternately be sealed using an intermediate symmetric key to the h/w TPM using the gSRK. The gSRK is a non-migratable key to ensure that it is not migrated with any vTPM when it is still in use by other vTPMs on the same platform. However, the vSRK is migratable because it is used only internally for storage protection. The vSRK in turn protects the storage of other binding keys which are used to encrypt (for storage) all other vTPM keys including vAIKs, signing keys and legacy keys. The vSRK could be a legacy key or a migratable binding key.

The vAIK$_{k,i}$ ($i^{th}$ vAIK of the $k^{th}$ vTPM) is a migratable signing key that is bound to the h/w TPM AIK using a non-migratable signing key (SK$_{k,i}$), i.e., the
h/w TPM AIK_{k,i} is used to sign the SK_{k,i}'s certificate which in turn is used to sign the vAIK_{k,i}'s certificate. This allows the vAIK to be migrated if required, unlike the case where the vAIK is directly signed by the h/w TPM AIK.

There is a special vAIK instance per vTPM (shown in red in the Figure 4.1(a)) that is used to sign keys that are never used (not even the public parts of these keys) outside the corresponding VM. We refer to this vAIK instance as the internal vAIK for the rest of this report. The corresponding SK and h/w TPM AIK are referred to as internal SK and internal AIK (also shown in red in the Figure 4.1(a)) respectively. This internal SK and internal AIK are common to all vTPM instances. The internal vAIK instance is used to create certificates for all binding, signing and legacy keys whose usage is only internal to the VM. Figure 4.1(a) shows two vTPM instances that share a common internal SK and internal h/w AIK. This is better than creating a new instance of the internal SK and internal AIK for each vTPM (VM) because the h/w TPM can hold only a limited number of keys at a time.

Figure 4.1(b) depicts the vTPM keys used outside the VM. Each vTPM could have several instances of vAIKs for attestation. Other vAIK instances that are used to sign keys used outside the VM may also exist. Such externally (outside the VM) used vAIKs are also bound to the h/w TPM via a non-migratable SK. However, a new SK and h/w TPM AIK is used for every vAIK that is used outside the VM to prevent tracing them back to the same platform. These keys are also protected by their vTPM’s vSRK and its binding keys.

4.1.2 Keys in a suspended VM’s vTPM

We now determine the subset of keys that may be present in the vTPM of a suspended VM. It is clear that any vAIK that is used for attestation should not be reused and hence, not migrated. Clearly, such keys should also not be part of a candidate vTPM (for migration) of a suspended VM. vAIKs can also be used to sign certificates for other keys. These vAIK certified keys can either be used only within the vTPM/VM for protected storage (confidentiality and/or integrity preservation via binding or sealing or signing) or can be used for communication outside the VM (for signing or encryption in sessions). In the latter case, key reuse can be detrimental to privacy and these keys must be deleted after a single use (and hence, before a VM is suspended, as a suspended VM has no active communication streams). In the former case, the vAIK and the keys whose certificates it signs can be migrated. Therefore, the only keys in the vTPM of a suspended VM include the vSRK, binding keys, keys that are used within the VM and the internal vAIK instance that is used to sign their certificates. These keys are used only in the VM and their migration does not allow tracking by a third party other than the source or the destination.

If the internal vAIK is signed using a h/w TPM AIK, it becomes non-migratable. It would have to be generated in the h/w TPM. It would also have to be regenerated on migration and all the certificates signed by it would have to be recreated. To
avoid this, here, we have introduced an intermediate non-migratable signing key (SK) which makes the internal vAIK migratable. This requires only recertifying the internal vAIK using a new signing key of the destination platform instead of recertifying all the keys signed by it.

vAIK generation

In the new hierarchy, vAIKs can be generated outside the h/w TPM but have to be signed inside the h/w TPM because the SK is a non-migratable key. They can either be generated randomly or could be structured. For example, the private part of the vAIK could be VM name concatenated with a timestamp and encrypted using a h/w TPM key. The corresponding public key can be derived mathematically using this private key. Such a key will be called a \textbf{structured key} for the rest of this discussion.

Including platform information in structured vAIKs which are public allows tracking the vAIK to the platform. Using VM information in them allows association of the vAIK to the VM. Using non-migratable h/w TPM keys in the derivation of public structured vAIKs allows proof of association with a TPM but this can also be achieved using the chain of trust from the vAIK to its AIK. For example, creating a structured internal vAIK using a non-migratable h/w TPM key could be used to ensure non-repudiation (via possible trace back to the TPM) during migration. \textit{However, using persistent h/w TPM keys (like SRK or EK) or VM (or platform) identification information (like the name) in the derivation of structured vAIKs that are used for attestation allows transaction linking. Use of structured keys as internal vAIKs derived using permanent h/w TPM keys allows a destination to link vTPMs that came from the same source platform (same physical platform rather than provider). Therefore, h/w TPM keys and VM/platform identification information should not be used in the generation of the vAIKs.}

4.1.3 vTPM key transfer during migration

During vTPM migration, the vTPM’s vSRK and its descendants can be transferred and used at the destination. The descendants include other binding keys, other internally used keys and the internal vAIK instance that is used to sign their certificates. \textit{However, this vAIK’s certificate is not migrated because this can allow tracing of different internal vAIKs to the same internal SK and internal AIK on the source. If the destination requires a proof of the chain of trust from the transferred vAIK to the h/w TPM, one could create a new SK and AIK at transfer time, use the new SK to create a new certificate for the vAIK before the transfer and then delete these keys after the transfer.}

At the destination, the migrated vTPM’s vSRK is first added as a child key of the destination platform’s gSRK. The destination itself has its \textit{internal SK} and its
A note on the integrity protection of vTPM keys

The vTPM keys are stored outside the TPM and are vulnerable to unauthorized access and modification. Since they are stored in encrypted form, their unauthorized modification leads to denial of the vTPM’s service for applications. Such modifications can be detected by concatenating each key with its hash before encrypting it and verifying this hash on retrieval. A more subtle problem would be the modification of the decrypted keys during communication between the vTPM and the h/w TPM. This can also be solved using hash verification.

Current implementations of software vTPMs use shared memory pages for communication between the vTPM and the vTPM manager in the trusted VMM. This secure channel prevents unauthorised modification of communication between the vTPM and the TPM through the vTPM manager. Since denial of service attacks by modification of vTPM keys on filesystem is possible even with the use of hashes, we do not use hashes to protect the integrity of vTPM keys in this work.

4.3 Protocol Outline

The high level outline of our proposed migration protocol is shown in Figure 4.2. We assume that an instance of the migration controller that handles migration
requests on each of the hosts (source and destination) coordinates the migration. Different vTPM designs may warrant different locations for the migration controller. For example, the migration controller could be part of the VMM in the vTPM virtualization solution in [27] but part of the Dom0 or the vTPM manager in Xen [30]. Hence, the exact location of the migration controller is implementation specific.

Figure 4.2: A VM-vTPM migration protocol outline

The protocol proceeds in four phases. Initially, the source and destination mutually authenticate each other and agree upon confidentiality and integrity preserving cryptographic mechanisms for protecting the rest of the transfer process. Next, the source sends an attestation request to the destination to ensure that the VM is migrated to a secure platform. Having ensured the authenticity and integrity of the destination platform, the source then locks the VM and vTPM and transfers them securely using the previously agreed upon cryptographic primitives. Then, the destination checks the integrity of the received VM and its vTPM. If no violations are detected, the destination imports the VM-vTPM pair (which is implementation specific) and sends an acknowledgment to the source on success. Finally, in the last phase, the source deletes the migrated VM and vTPM to prevent duplication and informs the destination that the migration is complete. The destination then resumes the newly received VM and its vTPM. The various phases of the protocol can be linked to a single session explicitly (using a session identifier) or implicitly (by ensuring that each phase depends any of the previous phases).
4.4 Detailed design

Figure 4.3 shows the sequence of exchanged messages for migrating a VM and its vTPM from one platform to another. In the following discussion, the content of each of these messages and their usage is explained.

4.4.1 Phase I: Secure session establishment

In this phase, the source and destination mutually authenticate each other and agree on cryptographic schemes to protect the confidentiality and integrity of the data exchange that follows. Host authentication uses public key certificates from a Certificate Authority (CA) and proof of knowledge of the corresponding private key. There are two choices to ensure confidentiality, namely, public key encryption or symmetric key encryption. Since symmetric key encryption is more efficient for bulk data transfers than public key encryption, it is common to exchange a symmetric key using public key cryptography and use it for bulk data encryption. The same approach is used in this migration protocol. Integrity can be ensured by the use of digital signatures, Hashed Message Authentication Codes (HMAC), checksums, etc. Again, since public key cryptography is resource intensive, we use HMACs instead of digital signatures for ensuring integrity.

Here, the TLS handshake protocol is used to derive the keys for data encryption and integrity preservation. This is immediately followed by the execution of the change cipher spec protocol. RSA is used to exchange the pre-master-secret. The handshake protocol results in two symmetric keys: $K_{enc}$ and $K_{mac}$ that are individually computed by the source and the destination using the information exchanged during the handshake. $K_{enc}$ is used for encryption (using symmetric ciphers like RC4, 128-bit AE3 or 3DES) while $K_{mac}$ is used for creating HMACs using SHA1.

Every message exchanged after this phase is encrypted using $K_{enc}$. The HMAC is concatenated to the encrypted message only before VM-vTPM transfer. The source and destination verify the HMAC of incoming messages (if they exist) and accept only messages with valid HMACs. Hence, the encryption key links all the protocol phases implicitly.
Figure 4.3: Our secure VM-vTPM migration protocol
4.4.2 Phase II: Remote attestation of the destination

Phase I establishes a secure session that protects the confidentiality and integrity of all messages exchanged in phase II. Since we trust the destination provider not to perform relay attacks, it is only necessary to ensure freshness in the attestation process to prevent replay of old configurations by a third party. Remote attestation protocols have been discussed in Section 2.2.6.

After the completion of phase I, the source creates a new nonce ($N_s$) and sends an attestation request along with it. The destination includes this nonce in its signature on the PCRs related to its Trusted Computing Base (TCB). The destination also generates a nonce ($N_d$) and sends it to the source which uses it to ensure freshness of the VM-vTPM transfer in the next phase. The source then checks the attestation reply. HMACs are used for integrity protection of the messages exchanged. On verifying the integrity of the destination platform, the source locks the VM and its vTPM to prevent further changes to them. The locking mechanism is implementation specific. It then sends a $SVR_{ATT\,OK}$ message to the destination. If any failures occur, a $SVR_{ATT\,FAILED}$ is sent instead.

4.4.3 Phase III: vTPM and VM transfer

All messages exchanged in this phase are also protected by the keys from the TLS handshake protocol. The exact semantics of vTPM and VM transfer depend upon the hypervisor and TPM virtualization solution. In all cases, the locked VM and its vTPM are concatenated and encrypted using $K_{enc}$. Then, the resulting message is concatenated with $N_d$ and the corresponding HMAC computed using $K_{mac}$ is added. The resulting encrypted data and corresponding HMAC are transferred to the destination. Nonce $N_d$ is used to prevent replay of the encrypted data to the destination. Only one VM-vTPM pair is transferred per session to prevent tracking (Section 3.5).

At the destination, upon receiving an encrypted blob, the HMAC is verified to ensure that the VM and vTPM were not modified during transit. If the verification fails, then a negative acknowledgment ($IMPORT\_FAILED$) is sent to the source and the received blob is deleted. If no modifications are detected, the VM and the vTPM are assigned their required resources. Then, the vTPM keys that are transferred in the process are imported using the process described in Section 4.1.3. On successful import, the destination platform sends an acknowledgment ($DONE$) to the source. This final message is encrypted using the encryption key from the TLS handshake. The nonces prevent replay attacks.

4.4.4 Phase IV: Deletion at the source

Upon receiving a DONE from the destination, the source deletes the VM and vTPM. However, if it receives a $IMPORT\_FAILED$ message instead, it does not
delete the VM or its vTPM. The source informs the destination that the migration is complete and the destination unlocks the VM and the vTPM. The confidentiality of all these messages is protected using the $K_{enc}$ key.

### 4.4.5 A note on VM attestation

The outlined protocol design assumes that the source and the migrating VM are trusted by the destination. This is reasonable in a setting where dynamic platform state measurement occurs ensuring that any malicious state changes to the source platform and the VM are detected and handled. Ideally, after the entire migration, the migrated VM should be able to report its new configuration to the destination on demand. This is meaningful only in a context where dynamic state measurements are enabled because otherwise, the VM could just replay its state before the migration.

### 4.5 Protocol evaluation

The above designed protocol is evaluated with respect to the security requirements (Section 3.5).

a. **Independence from the integrity measurement mechanism**: The above protocol is not tied to any specific integrity measurement mechanism (binary measurement or property based measurement). It only defines the communication sequence between the source and destination without including any of these details. It is also independent of the VMM.

b. **Authenticated data transfer between secure platforms**: MITM attacks are prevented by the establishment of a secure (authenticated and confidential) channel from the source to the destination using TLS. Bait and switch attacks are prevented by connecting the different protocol phases using the keys from the first phase. Migration of a vTPM to an insecure platform is prevented by destination platform attestation in Phase II prior to the actual transfer.

c. **Secure VM-vTPM association**: A VM is associated with its vTPM by transferring both of them together in the same message protected by a single HMAC. Thus, the VM-vTPM association is implicit in this protocol.

d. **Confidential data transfer**: The secure channel establishment results in a symmetric encryption key known only to the source and the destination. The use of a symmetric key to encrypt the VM and its vTPM prior to actual transfer ensures confidentiality.

e. **Integrity preserving data transfer**: The secure channel establishment results in a symmetric key that can be used to create HMACs and is known only to the source and the destination. The use of HMACs allows detection of any modification to the VM or vTPM during transit. This follows from the property of HMACs which makes it impossible for an attacker to create a valid HMAC for a modified VM or vTPM without knowing the symmetric key.
f. **Replay resistance**: Although an attacker cannot access the content of a VM or vTPM because of encrypted transfer, it may be possible for him to record and replay the same VM and vTPM to the destination at a later time. This is prevented by using nonces at every step in the protocol making it impossible for an attacker to replay an old encrypted message without being detected at the destination.

g. **Source non-repudiation**: The use of public key certificates in secure channel establishment ensures that only an entity aware of the source’s private key can successfully participate in the negotiation. Hence, only such an entity can derive the symmetric keys that are used subsequently in the protocol. Ideally, only the source provider must be aware of the private key corresponding to the source’s X.509 certificate. This ensures non-repudiation.

h. **Atomicity of the transfer**: Atomicity encompasses prevention of duplicate generation and data loss. Generation of new copies of a VM and its vTPM is prevented by deleting the VM and its vTPM from the source in case of a successful transfer and deleting it from the destination in the event of a failure. Recovery from failed migration attempts is ensured by keeping a copy of the VM and its vTPM at the source until successful migration is confirmed.

i. **Privacy preserving data transfer**: In the designed protocol, only one VM migration occurs per session. The entire communication sequence is encrypted and prevents an attacker from learning about any of the attestation keys (and hence, the host platforms) used in it. Each session uses a new set of keys for attestation. The migrated vTPM does not retain any keys that it uses publicly across transactions. This prevents the source from tracking the migrated vTPM using its public keys. However, a source may still be able to recognize a vTPM that it hosted earlier using its internal keys (when it is migrated back to it) but we assume this is not a major threat in intra-cloud scenarios.
Chapter 5

Implementation

The secure migration protocol described in the previous chapter has been implemented in the Xen hypervisor (version 4.0.2-rc1). The purpose of the implementation was not only to develop a proof-of-concept but also to perform a performance evaluation. This chapter describes the different aspects involved in the implementation including certain implementation choices, the setup and the actual protocol implementation in detail.

5.1 The implementation choices

The implementation required the choice of a hypervisor that supports VM migration and a vTPM architecture design. Since most hypervisors support VM migration, it was necessary to check if the existing VM migration (and the vTPM migration if it exists) was flexible enough for reuse. These implementation decisions and the rationale behind them are elucidated below.

5.1.1 The hypervisor

We chose the Xen hypervisor because it is open-source, has a good support framework and is popular in research circles. Although Xen supports VM migration (Section 3.2), reusing this implementation does not offer flexibility in the timing of the restoration of the VM (which is required by our protocol because we do not resume the VM immediately after its transfer to the destination). Finally, Xen’s use of Python for implementation would make performance analysis of our protocol complicated if we reused it. Hence, although we used the Xen hypervisor, we did not directly use its existing support for VM migration. Instead, we made use of Xen’s VM suspension and resumption capability which allows saving the state of the VM to a file to implement VM migration. Xen’s existing vTPM migration protocol and its deficiencies have already been discussed in section 3.3.3.
5.1.2 The vTPM architecture

Among the existing designs [Section 2.3], a proof-of-concept implementation was available only for the software TPMs described in [26]. Their implementation has been integrated into Xen [Section 2.3.1]. Currently, work to integrate the TPM Emulator from here into this framework is in progress which made it unusable for the current project. Since the implementation of a vTPM architecture was beyond the scope of this project, we used an alternate architecture consisting of each vTPM instance running as a process within its own VM instead of in Dom0. We also reuse the TPM emulator as the vTPM. This imposes some additional security requirements on the TPM emulator and is discussed in the TPM emulator’s setup as a vTPM [Section 5.2.4]. A brief discussion of the impact of running the vTPM instance inside the VM instead of in Dom0 is presented below.

vTPM inside VM vs. vTPM in Dom0

There are several tradeoffs between running the vTPM inside the VM versus running it in Dom0 of Xen. Running the vTPMs in Dom0 is more secure than running the vTPM inside the VM itself. Establishing a chain of trust dynamically from boot to the VM can mitigate this to an extent. Besides, vTPMs in Dom0 process space also require a separate process migration implementation for their transfer and re-association with their corresponding VMs at the destination. Running the vTPM inside the VM is more efficient than running it in Dom0 if the dynamic VM state measurement module is inside the VM and vice-versa. When the vTPM is run inside the VM, VM suspension and migration automatically transfers the vTPM to the destination which is easier and perhaps slightly more efficient than if the vTPM is run in Dom0. It also obviates the need for any re-associations between the VM and its vTPM post-migration and hence, for any identity clash resolution procedures for the vTPMs. Running the vTPM inside the VM also makes boot time measurements impossible.

An additional aspect of running the vTPM inside the VM instead of in Dom0 that is relevant for migration performance, is that the vTPM’s storage which would have to be decrypted before transfer to the destination is on the VM’s filesystem instead of the Dom0’s filesystem. This requires mounting the VM’s filesystem before transfer, decrypting the non-volatile storage protection key and then transferring the VM’s filesystem. Similarly, at the destination, the arriving VM’s filesystem is mounted and the key is bound to the destination’s h/w TPM before the VM is restored.

While we understand that our alternate architecture is less secure but potentially more efficient, a performance analysis even with the modified architecture is useful because, given a certain hardware setup, it allows us to study the effect of using different ciphers and VM RAM sizes on the performance of our implemented migration protocol.
5.2 The implementation setup

The hardware setup used for the implementation and the software installations involved in the development phase of the protocol are discussed below.

5.2.1 The hardware

The hardware used for the implementation consists of two Lenovo Thinkpad laptops (T60 and T60p). Each machine is equipped with an ATmel TPM (chip version 1.2.11.5), an Intel Dual Core processor (the Lenovo T60 with a 2.0 GHz processor and the Lenovo T60p with a 2.16 GHz processor) and 2GB RAM. Both run the Xen hypervisor (version 4.0.2-rc1) and are used to host VMs. A third Toshiba Satellite Pro laptop with 3GB RAM and Intel Centrino processor (running Xen 4.0.1-rc5) was used as an NFS server to host the shared storage. This shared storage was used to hold the disk images and the configuration files of the VMs. All the hosts are connected via a 1GB Ethernet LAN.

5.2.2 Xen installation

The Xen hypervisor consists of a custom kernel that can be compiled into any compatible Linux system. The version of Xen used for this project is 4.0.2-rc1 and the kernel version used is 2.6.32.21. The Xen kernel was compiled on an Ubuntu 10.04 Lucid Lynx installation. Ubuntu 10.04 by default uses an ext4 filesystem and grub version 2. But since the Xen kernel does not support ext4 filesystems, an ext3 filesystem was used for the original Ubuntu installation. The procedure to install the latest release candidate of Xen is shown in Figure 5.1 (as described partially in http://bderzhavets.wordpress.com/2010/04/). This procedure automatically downloads the latest Dom0 kernel and compiles it with default configuration options.

```
# apt-get install libcurl4-openssl-dev xserver-xorg-dev mercurial git
# apt-get install libncurses5-dev dpkg-dev debhelper ia32-libs bridge-utils bison flex
# cd /usr/src
# hg clone http://xenbits.xensource.com/xen-4.0-testing.hg
# cd xen-4.0-testing
# make world
# make install
# cat /usr/lib/python2.6/site-packages /use/lib/python2.6/dist-packages/p.pth
Add the following line to /etc/defaults/rcamilone /proc/xen sends defaults 0 0
Modify /etc/xen/xen-config.sxp to use NAT for the VMs
# apt-get remove network-manager
# update-grub
```

Figure 5.1: Xen installation procedure

VMs can be created by either using a common DomU kernel on the Dom0 filesystem for all VMs or by using Xen-tools from http://www.xen-tools.org/software/xen-tools/releases.html (installed as shown in Figure 5.2) that allows the DomU kernel to reside on the DomU filesystem and uses pygrub (similar to grub) to boot.
the VM. In this project, we use Ubuntu 10.04 Lucid Lynx kernels (version 2.6.32.24-pae-generic) for the VMs with each VM’s kernel on its own filesystem. We use xen-tools-4.2-beta1 with some modifications (because it did not support installation of Lucid Lynx kernels directly unlike the latest version.)

<table>
<thead>
<tr>
<th>VM creation using xen-tools-X</th>
</tr>
</thead>
<tbody>
<tr>
<td>Download the zipped version of latest version of xen-tools and extract it.</td>
</tr>
<tr>
<td># cd xen-tools-X</td>
</tr>
<tr>
<td># make install</td>
</tr>
<tr>
<td>Modify the /etc/xen-tools/xen-tools.conf to create the bootable disk file of the chosen distribution.</td>
</tr>
<tr>
<td># xen-create-image –hostname=</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Modifications to xen-tools-4.2-beta1 for Lucid Lynx kernels</th>
</tr>
</thead>
<tbody>
<tr>
<td># cp -r /usr/lib/xen-tools/edgy.d /usr/lib/xen-tools/lucid.d</td>
</tr>
<tr>
<td># ln -s /usr/share/debootstrap/scripts/gutsy /usr/share/debootstrap/scripts/lucid</td>
</tr>
</tbody>
</table>

Figure 5.2: VM creation procedure

5.2.3 The hardware TPM setup

In order to be able to access the TPM, a number of configuration settings have to be enabled in the BIOS and the kernel. First, the TPM chip must be enabled (state=ENABLED) in the BIOS security options. It is then reset or cleared from the BIOS. The installed Xen kernel does not have the TPM driver compiled by default. It is therefore, re-compiled after enabling the default TPM 1.2 Linux driver (compatible with TPM Interface Specification 1.2 or TPM TIS) under character devices and the pseudo-security file system option under file systems as modules. Then, an open-source TCG software stack for Linux called Trousers ([http://trousers.sourceforge.net/](http://trousers.sourceforge.net/)), a set of command-line tools to interact with the TPM called tpm-tools and finally, the development library for TPM applications called libtspi-dev are installed. The tpm-tools package provides utilities (using the command tpm_takeownership) to take ownership of the TPM and set the owner password and the SRK password. Further management of these secrets is also done using the same package (using the command tpm_changeownerauth). Finally, the ’/etc/modules’ file is modified to load the TPM driver and the Trousers modules on boot.

5.2.4 The TPM emulator (vTPM) setup

The TPM emulator (Section 2.3.6) installation requires cmake (version 2.8.X) which can be obtained from [http://www.cmake.org/files/v2.8/](http://www.cmake.org/files/v2.8/) and the emulator itself can be downloaded from [http://download.berlios.de/tpm-emulator](http://download.berlios.de/tpm-emulator). We use cmake version 2.8.2 and TPM emulator version 0.7. Its installation procedure is shown in Figure 5.3.

The TPM emulator must satisfy additional security properties before it can be used as a vTPM. Its storage file must be protected against unauthorized access.
The implementation setup

<table>
<thead>
<tr>
<th>INSTALLATION OF THE TPM emulator</th>
</tr>
</thead>
<tbody>
<tr>
<td># apt-get install linux-headers-<code>uname -r</code> linux-source-2.6.3X lib64-dev libgmp3-dev</td>
</tr>
<tr>
<td>Download the zipped version of cmake and extract it.</td>
</tr>
<tr>
<td>Download the zipped version of TPM emulator and extract it.</td>
</tr>
<tr>
<td># cd tpm_emulator-0.X</td>
</tr>
<tr>
<td># mkdir build</td>
</tr>
<tr>
<td># cd build</td>
</tr>
<tr>
<td># path-to-cmake-2.8.X/bin/cmake ../</td>
</tr>
<tr>
<td># make</td>
</tr>
<tr>
<td># make install</td>
</tr>
<tr>
<td># depmod -a</td>
</tr>
<tr>
<td>Reboot.</td>
</tr>
</tbody>
</table>

Figure 5.3: TPM emulator installation procedure

Since the h/w TPM only allows binding data of size of limited size (256 bytes) at a time, protecting the storage file (which can be of arbitrary size) with it would be inefficient. Instead a key of appropriate size (max. 256 bytes) is used to encrypt the storage file using DES in CFB mode (using source from: http://www.codealias.info/technotes/des_encryption_using_openssl_a_simple_example) and this key is in turn bound to the h/w TPM via the gSRK.

Running the vTPM inside the VM implies that boot time measurements are not feasible any more. Also, since each vTPM’s non-volatile storage is bound to the h/w TPM via the gSRK, for each access to its storage, the vTPM would have to communicate with h/w TPM via Dom0. But since the emulator only writes the non-volatile storage when shutdown, the h/w TPM (and hence Dom0) needs to be contacted only during startup and shutdown of the emulator. We have implemented a custom VM startup script that starts a server on Dom0 which listens to requests for binding and unbinding files to the h/w TPM using the gSRK. It also measures the kernel image of the VM. A custom vTPM start up script inside the VM connects to this server to unbind a file containing the symmetric key used to encrypt the vTPM storage file. It decrypts the emulator’s storage files and starts the vTPM (emulator). Then it extends the PCR2 (although it could be any other PCR) of the vTPM with the measurement values that it obtained from the Dom0 during startup. At the moment, the only value being measured is the kernel image of the VM. A similar shutdown script to re-encrypt the changed storage file can also be written.

5.2.5 Trusted boot

TrustedGrub (http://sourceforge.net/projects/trustedgrub/) is used to measure and extend PCRs of the h/w TPM on booting Dom0. Since Ubuntu 10.04 comes with GRUB version2, it is first downgraded to grub version 0.9 (http://fordflux.com/blog/linux/downgrade-grub-2/). Then, GRUB version 0.9 is replaced by trusted GRUB version 1.1.4 using instructions downloaded with the installer (Figure 5.4). In this work, all measurements are restricted to boot time and no dynamic measurements are made.

The ATMEL TPM on the two Lenovo laptops had 24 PCRs. The summary of
PCRs extended at boot can be found in the README document of TrustedGRUB and at the Lenovo Thinkpad Wiki (http://www.thinkwiki.org/wiki/Embedded_Security_Subsystem). This latter documentation is for old Thinkpads. In general, on boot, the BIOS touches PCRs 0-7 and leaves the PCRs 8-15 untouched for the user. After booting using TrustedGRUB, PCR 4 contains information about the MBR and stage 1 of GRUB, PCR 8 and PCR 9 contain information about GRUB stage 2 (part 1 and part 2 respectively), PCR 12 contains information about all command line arguments from menu.lst and PCR 14 contains information about all files actually loaded (Linux kernel, initrd, etc.).

5.2.6 The shared storage setup

The shared storage consists of an NFS server that is set up by enabling the NFS server option during kernel compilation. Access rights to the shared directory are configured using the ”/etc/exports” file. The migration source and destination are given read and write access in synchronous mode. A fixed folder containing configuration files, disk files and swap files of several VMs is exported. The NFS client is enabled by default in the installed Xen kernel.

5.3 The implemented insecure protocol

The implementation of the insecure protocol is shown in Figure 5.5. The connection setup uses simple UNIX sockets that allow a maximum send/receive buffer size of 16KB.

5.3.1 VM suspension and restoration

The Xen hypervisor allows saving VM state for suspension in a file and resumption of the VM using the same file. This suspension and resumption is implemented partially in Python in Xen. The VM is suspended on the host and the saved state file is transferred to the destination where it is used to restore the VM. The command-line interface provided by xen-tools is used for this purpose.
5.3.2 VM migration with or without disk transfer

Two versions of the secure migration protocol have been implemented. The first version includes the transfer of the disk files (configuration file, filesystem and swap images) of the VM along with its RAM image during migration. The second version assumes that the configuration file, the filesystem and swap images of the migrating VM are on a common shared storage which is accessible to both the source and the destination of the migration. Therefore, it transfers only the RAM image of the VM.

5.4 The implemented secure migration protocol

The following subsections contain details of the implementation of different phases of the protocol. The language used for the implementation is C. Libraries used include libtspi-dev (version 1.1), libcurl4-dev (version 4.1.1) and openssl (version 0.9.8o).

5.4.1 Secure connection setup

OpenSSL is used to create a secure connection between the migration source and the destination. It is compiled with compression using the zlib library disabled. A new self-signed CA certificate is installed on both the source and the destination. It is used to sign their public key certificates that are used for authentication during connection setup. The ciphers used for communication are 128-bit AES, RC4 and 3DES for encryption and SHA1 for creating HMACs. The use of OpenSSL for communication appends HMACs for integrity checks to all messages although this is not necessary. However, since OpenSSL is the current standard for secure communication, it has been used in the implementation despite generation of HMACS for all messages rather than just when required. Similarly, the use of OpenSSL also obviates the need for nonces $N_{s2}$ and $N_{d2}$ (as shown in Figure 4.3) and has been
removed from the implementation. OpenSSL sockets allow a maximum buffer size of 16KB and therefore, all large data transfers are done in steps of 16KB. The implemented protocol is depicted in Figure 5.6.

Figure 5.6: The implemented secure migration protocol

Initially, the source sends a hello message containing the version of TLS it supports, some random data ($N_s$) and a cipher specification. Here, the chosen cipher specification includes RSA to exchange the pre-master-secret which is used to derive the session keys, 128bit-AES or RC4 or 3DES for encryption and SHA1 for HMAC creation. The destination responds with a corresponding hello message. TLS allows resuming old sessions but attestation has to be performed every time on resuming a session. We have implemented single instance migration servers and do not face this problem currently. But it is possible in practice for the destination to force a
The implemented secure migration protocol

new session every time by issuing a new session id.

The server then sends its public X.509 certificate for authentication. To authenticate the migration source, the destination requests a RSA public key certificate signed using an RSA key by a CA and a list of CAs that it trusts. The source verifies the destination’s certificate and responds by sending its public key certificate. It then computes a pre-master-secret and encrypts it using the destination’s public key \((K_{Pub})\). This pre-master-secret is used along with a Pseudo Random Function \((PRF)\) to derive the master-secret (Figure 5.7) which is in turn used to obtain the keys for encryption and creation of HMACs. The source also sends a signature on all the previous messages exchanged using the private key corresponding to its public key certificate as proof-of-knowledge of its private key. Finally, the change cipher spec protocol is executed to switch to communication using the new security configuration.

**Figure 5.7:** Calculation of the master-secret and the session keys in the TLS protocol

<table>
<thead>
<tr>
<th>TLS master secret generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>master-secret = PRF(pre-master-secret, string(master secret), N_s + N_d)</td>
</tr>
<tr>
<td>key-block = PRF(master-secret, string(key expansion), destination-random + source-random)</td>
</tr>
</tbody>
</table>

Partition key-block into:

a. Source encryption key, MAC secret
b. Destination encryption key, MAC secret

5.4.2 Attestation

The framework for remote attestation consisting of a privacy CA and valid PCR values is not yet available. A test PCA is currently available at [http://www.privacyca.com](http://www.privacyca.com). There are also source files available for obtaining an AIK, signing PCRs using this AIK and verifying such a signature. AIKs can be obtained by either presenting a valid EK that can be verified using the TPM vendor’s certificate to the test PCA or simply presenting a fake EK certificate with the correct RSA modulus size without verification. We use the latter approach to create AIKs because the TSS stack was unable to read the EK certificate of the TPMs on the Lenovo laptops correctly. This and other source files from [http://www.privacyca.com](http://www.privacyca.com) have been modified appropriately for usage during migration. The changes include writing output to appropriate files, specification of passwords for SRK and EK inline instead of in pop-ups (because they did not work with this version of Trousers).

PCR values are verified against static values recorded during the first run. The AIK certificate is verified by installing the certificates at [http://privacyca.com/cert_root.html](http://privacyca.com/cert_root.html) and [http://privacyca.com/cert_level0.html](http://privacyca.com/cert_level0.html) as trusted CA certificates.
5.4.3 File locking

Most Linux systems only support advisory file locking, i.e., even if a process obtains an exclusive lock on a file, the kernel does not prevent another process from writing to the file. It is assumed that all processes check for the presence of a lock instead of enforcing it through the kernel. Mandatory locking mechanisms are available in custom kernels but for the purpose of the project we use the available advisory locking mechanisms provided by Linux.

VM suspension and resumption is done using Xen’s suspension and resumption features as described for the insecure protocol. Finally, two versions of the secure protocol corresponding to the two versions of the insecure protocol have been implemented. One version transfers the VM’s file system along with its RAM image from the source to the destination and one which transfers only the VM RAM image assuming that the source and the destination have access to a shared storage that hosts the VM’s filesystem.

5.5 Known issues

Since the disk file of the VM is modified agnostic to the VM’s operating system during migration (vTPM’s storage file encryption key file is decrypted after suspending the VM), resuming the VM at the destination shows an old VM filesystem state. This problem is orthogonal to our implementation. However, re-migrating the VM without shutting it down works correctly. On shutting down a newly migrated VM, the vTPM’s storage encryption key file gets corrupted and further migration attempts fail. We believe that this file corruption is also due to the inconsistent filesystem view of a resumed VM. Another problem is the shared storage state not being updated quickly enough after the migration source decrypts the vTPM storage file encryption key resulting in the destination not finding decrypted key file after the migration for small VM RAM sizes. This also led to intermittent migration failures.
Chapter 6

Performance evaluation

One of the most important benefits of virtualization that is widely used is VM migration. Performance of VM migration is critical to most of its applications. As a result, live migration is the more commonly used than suspended VM migration and improving the efficiency of live migration is a hot topic in contemporary research [34], [37]. Analogously, the widespread adoption of any secure migration protocol hinges on its performance as compared to its insecure counterpart. In the following sections, a performance evaluation of the secure migration protocol and its corresponding insecure version is presented. For the rest of this chapter, we refer to the migration source and destination as the client and the server respectively.

6.1 Methodology

In this section, we discuss our choice of performance metrics and profiling tool followed by the profiling strategy used for the analysis.

6.1.1 Performance metrics

We identified the following performance metrics:

a. **System time (for an operation/protocol phase/process to complete):**
   This is the time for an operation or protocol phase or an entire process to complete as measured using the system clock. For instance, total time for VM migration refers to the total time measured from when the client is started to when the server ends.

b. **CPU time:** The CPU time consumed by a process is measured using a profiler. Unlike total system time for an operation, CPU time consumed by an process represents the actual load due to that process on the CPU.

c. **Memory usage:** The memory requirements of a process is important because it can directly affect the efficiency of co-existing processes. For example, memory usage patterns of the client and the server during migration is important because it affects the performance of applications in other co-existing VMs. By memory usage we mean memory allocations on the heap.
d. **Network throughput**: Network throughput refers to the average amount of data that is transferred to the communication channel in unit time. It is measured in terms of bits per second. Like memory usage, it is important because it can affect the performance of other co-existing networking applications (in other VMs or even other hosts). It is noted that network throughput strongly depends upon any data compression techniques that may be used.

_In this work, we used only the system time and/or CPU time metrics to compare the secure and the insecure protocol implementations. We intend to perform memory and network measurements as part of future work._

### 6.1.2 Measured protocol operations

Our secure protocol consists of several phases as described in Figure 4.3. Each of these phases has a number of operations. These operations have been implemented using a number of routines (functions). Some of these operations are common to the secure and insecure versions of the protocol. The above chosen metrics can also be measured at different granularities, namely, for each process or phase or operation or each routine (function) in the implementation. The choice of granularity is application specific. We measured each of the chosen metrics namely, system time and cpu time for the following phases and operations.

a. **Connection establishment**: For the insecure protocol, we measured the connection setup time using simple sockets. For the secure protocol, we measured the total time for the SSL handshake and the server attestation to complete using the system clock. We also measure the corresponding CPU time. This allows us to evaluate the overhead due to secure connection establishment.

b. **VM suspension and resumption**: We measured only the total time for VM suspension and migration using the system clock. It was not possible to measure the CPU time for suspension and resumption because it is partially implemented in Python.

c. **End-to-end migration**: We measured the total CPU time consumed individually by the client and the server since it represents the actual load on them. Only the total (system) time that elapsed between the time at which the client starts execution and the server execution ends (referred to as net migration time or total migration time) was measured because this represents the visible VM downtime to the end user. Individual client and server execution times measured using the system clock are neither indicative of load (which is given by their CPU consumption) nor VM downtime.

d. **Individual operations**: We measured the CPU time consumed by individual operations (such as encryption, hashing, etc.) in order to identify the bottlenecks. The total time (using the system clock) spent in individual routines was not collected because this would not help detect bottlenecks.
6.1.3 Measurement technique

The performance metrics can be measured in many different ways. System time for a certain operation can be measured using existing Linux functions like `gettimeofday()` at the start and end of the function. CPU time can be measured using theoretical computations of resource consumption (CPU cycles/time, bytes of memory, etc.) or using profiling tools. We chose to use the latter approach to measure CPU time as it allows us to vary a number of protocol parameters (cipher, VM RAM size, etc.) more efficiently. Profiling tools can be classified depending on various criteria including whether they are instrumented into code for use or not, whether they are enabled at compile time (like google-perf tools using code instrumentation), link time (gprof) or run time (google-perf tools using environment variables). In [38], a brief survey of different Linux profiling tools is presented. In the following discussion, we describe the choice of a profiling tool, how it works and its output.

Choice of a Profiling tool

There are a number of commercial (Intel VTune) and open source profiling tools (google-perf tools, gprof) that can be used to evaluate the cpu and heap usage patterns of programs. The Intel VTune tool is capable of reporting the number of CPU cycles per operation and has been used to evaluate OpenSSL performance [39]. But this tool was not compatible with the Xen kernel we used. The use of 'gprof' for CPU performance analysis and google-perf tools' heap profiler resulted in system crashes. The google-perf tools' CPU profiler was used to evaluate CPU usage of the client and the server during secure and insecure VM migration.

The google-perf tools

The google-perf tools is an open source set of performance evaluation utilities. It was installed using instructions from [http://google-perftools.googlecode.com/svn/tags/perftools-1.6/INSTALL](http://google-perftools.googlecode.com/svn/tags/perftools-1.6/INSTALL). The google-perf tools' CPU profiler works by sampling the call stack at regular intervals. The tool raises an interrupt periodically using the Linux interval timer (with ITIMER_PROF) to collect samples. This allows collection of samples not only when the profiled program is executing but also when the CPU is being used on its behalf. The default interval between collection of samples of 10ms was used for the purpose of this project and could not be decreased due to memory considerations.

After installation, the profiler is instrumented into code and compiled with appropriate linker directives (include the `-lprofiler` directive during compilation). A program can be instrumented with code for profiling by inserting `CPU_PprofilerStart(profile_name)` and `CPU_PprofilerStop()` around the parts of code to be profiled. When the program is executed, a profile with the specified name is created in the current directory of the program.

---

1One may have to run the command 'ldconfig' as root to complete installation
Profiles generated by the google-perf tools can be analyzed to yield graphical or textual outputs. In either case, the information includes the number of samples collected in each routine of the program. Since the sampling rate is 1 per 10ms, the amount of time spent in each routine is the number of samples collected in that routine multiplied by 10ms. In the textual output, the routines are ranked according to their sample count. Relative CPU time utilization of routines is also available. In the graphical output, routines are displayed in a call-graph that gives information about the control flow in the program. The size of each node corresponds to its relative sample count.

Profile collection

We use the setup described in Section 5.2 to measure the performance parameters. VMs with a 1GB disk space (filesystem), 128MB swap space and different sizes of RAM, namely, 128MB, 256MB, 512MB, 768MB and 1GB were used for the evaluation. Larger filesystem sizes could not be used due to memory constraints. Further, the secure and insecure protocols are evaluated in two contexts as described in Section 5.3.2.

During insecure migration, a VM with a certain RAM size was migrated six times - thrice with the Lenovo T60 as the source and thrice with the Lenovo T60p as the source in order to obtain three client profiles and three server profiles on the Lenovo T60p host. Further, this was repeated for five different sizes of RAM. The secure migration protocol was also executed similarly and the CPU profiles (server and client) for three different ciphers (128 bit AES in CBC mode, RC4 and 3DES in CBC mode) were collected. Finally, the client and server profiles obtained on the Lenovo T60p host were analyzed. All the timing information was used to obtain 95 percent confidence intervals for the measured metrics of the mentioned protocol phases.

The overhead experienced during secure connection setup was measured by modifying the secure client and server to establish an SSL connection, perform server attestation and close the connection. This was also done six times to obtain three client profiles and three server profiles on the Lenovo T60p host for analysis.

Limitations

The use of profiling tools makes our analysis dependent on the hardware we use. The non-standard nature of our virtualization hardware (described in section 5.2.1) does not give us a context that is directly comparable to existing results in typical VM migration application scenarios. Since the hardware in typical usage scenarios is expected to perform better than our setup, performance in more practical scenarios is expected to be better. Another drawback with the hardware used is that the source and the destination are not identical. Despite the hardware not being representative of the real world virtualization architectures, a performance analysis still provides useful insight into aspects like the impact of different ciphers and dif-
6.2 Analysis of results

In this section, we present a comparison between the performance of the secure migration protocol and its insecure counterpart. First, we discuss the CPU time and the system time required for the secure connection establishment followed by the total system time consumed by VM suspension and resumption. Then, we compare end-to-end migration using the secure migration protocol to end-to-end migration using the insecure protocol. The CPU time distribution among different operations has also been discussed.

6.2.1 Connection establishment

The system time and CPU time metrics for the connection establishment phase are as shown in Table 6.1. In case of the insecure protocol, the connection establishment completes in less than a second and the CPU profiler records no CPU time consumption. In case of the secure protocol, the time for the connection setup is consistent and is about $\approx 8\text{s}$ (measured using system time). But the CPU-profiler records less than $\approx 0.05\text{s}$ (50ms) of CPU time for the client and less than $\approx 0.08\text{s}$ (80ms) of CPU time for the server irrespective of the cipher used. On profiling just the creation of an AIK key during attestation, it is observed (across three trials) that a small amount of CPU time ($\approx 0.05\text{s}$ (50ms)) is consumed but the entire AIK key creation takes $\approx 5\text{s}$ of system time. This preliminary investigation suggests that the creation of the AIK key is the bottleneck in the secure connection setup. Most of the real time is likely to be spent waiting for a response from the PrivacyCA or at the TPM. Further investigation is needed to confirm this hypothesis.

Table 6.1: Secure connection establishment time using different ciphers

<table>
<thead>
<tr>
<th>Cipher</th>
<th>Client CPU time</th>
<th>Server CPU time</th>
<th>Total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insecure</td>
<td>NA*</td>
<td>NA*</td>
<td>&lt;1</td>
</tr>
<tr>
<td>128-bit AES</td>
<td>0.03</td>
<td>0.07</td>
<td>8.33</td>
</tr>
<tr>
<td>RC4</td>
<td>0.02</td>
<td>0.07</td>
<td>8</td>
</tr>
<tr>
<td>3DES</td>
<td>0.03</td>
<td>0.07</td>
<td>7.66</td>
</tr>
</tbody>
</table>

* The profiler was unable to collect samples during this execution.
6.2.2 VM suspension and resumption time

The VM suspension and resumption interface in Xen is partially implemented in Python and was therefore, not profiled. However, this is common to the secure and insecure versions of the protocol. The time taken to suspend and resume VMs with differently sized RAM images during migration using different ciphers was recorded and is shown in Table 6.2. This data was recorded during migration without disk transfer and averaged over three trials. The suspension and resumption time for a given VM RAM size was fairly consistent irrespective of whether the disk image was on the local machine or on a remote NFS (hosted by the Toshiba Satellite Pro.).

(a) VM suspension time on Lenovo T60p

<table>
<thead>
<tr>
<th>VM RAM size (MB)</th>
<th>Suspension time on client (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Insecure</td>
</tr>
<tr>
<td>128</td>
<td>5.67</td>
</tr>
<tr>
<td>256</td>
<td>11</td>
</tr>
<tr>
<td>512</td>
<td>20.33</td>
</tr>
<tr>
<td>768</td>
<td>30</td>
</tr>
<tr>
<td>1024</td>
<td>38.33</td>
</tr>
</tbody>
</table>

(b) VM resumption time on Lenovo T60p

<table>
<thead>
<tr>
<th>VM RAM size (MB)</th>
<th>Resumption time on server (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Insecure</td>
</tr>
<tr>
<td>128</td>
<td>3</td>
</tr>
<tr>
<td>256</td>
<td>3.33</td>
</tr>
<tr>
<td>512</td>
<td>5.33</td>
</tr>
<tr>
<td>768</td>
<td>15.33</td>
</tr>
<tr>
<td>1024</td>
<td>24</td>
</tr>
</tbody>
</table>

Table 6.2: VM suspension and resumption time on Lenovo T60p during VM migration using different ciphers

The time for suspension increases almost linearly with the VM RAM size. The time taken for resumption also increases with increase in VM RAM size although the increase does not seem linear. This maybe due changes (decrease) in the available memory caused during VM restoration.

6.2.3 End-to-end migration

Individual client and server execution time

The individual CPU time of the client and the server is a better indicator of efficiency than the individual system times for client and server execution. This is because the individual system times include periods of waiting for I/O during which the CPU is idle and available for use by other processes. In practice, such migration modules (client and server) will be multi-threaded to support migration of several VMs simultaneously and therefore, will be able use the CPU more efficiently. Hence, individual total system times for client and server execution were not measured.
Table 6.3: Average CPU time for end-to-end VM migration without disk transfer using different ciphers

Table 6.4: Average CPU time for end-to-end VM migration with disk transfer using different ciphers

Table 6.3 shows the CPU time consumed during VM migration without disk transfer by the client and server. Figure 6.1 shows how this CPU time is distributed among various operations (read, write, cryptographic operations). The figure also shows 95 percent confidence intervals for the total CPU time in every case. Confidence intervals for individual operations have been omitted for clarity.
Chapter 6

(a) Client CPU time consumption

(b) Server CPU time consumption

Figure 6.1: Client and server CPU time distribution between different operations for different VM RAM sizes and ciphers during VM migration without disk transfer (Confidence intervals are omitted for individual operations and shown only for the total CPU time consumed to preserve clarity.)

The CPU time for the client and the server using the insecure protocol is directly proportional to the VM RAM size. Also, the insecure server execution consumes more CPU time than the insecure client execution. The distribution of CPU time between different operations indicates that read and write operations consume most of the client and the server CPU time in case of insecure VM migration.

The secure protocol consumes more CPU time both at the server and the client. The client and server CPU time consumption as well as the CPU time for individual
**Analysis of results**

(a) Client CPU time consumption

(b) Server CPU time consumption

Figure 6.2: Client and server CPU time distribution between different operations for different VM RAM sizes and ciphers during VM migration with disk transfer. (Confidence intervals are omitted for individual operations and shown only for the total CPU time consumed to preserve clarity.)

Operations are directly proportional to the VM RAM size. The profiles for secure server and secure client reveal that the time spent in cryptographic operations and in read and write system calls accounts for more than 90 percent of the overall CPU time. A significant portion of this time is spent in the cryptographic functions. The CPU time for secure migration using RC4 is over 50 percent higher compared to the CPU time using the insecure protocol, for both the client and the server. The client and server CPU time for secure migration using 128-bit AES is nearly twice the respective values for insecure migration. Actual distribution of the time spent in each operation (encryption, HMAC using SHA, read and write) is fairly consistent.
across multiple migration runs for a given cipher and VM RAM size. It is interesting to note that the amount of time spent in the read and write system calls is less in the secure version (with any cipher). We think that this maybe due to OpenSSL’s optimized read and write.

(a) Total migration time without disk transfer (Lenovo T60p as client)

(b) Total migration time without disk transfer (Lenovo T60p as server)

Figure 6.3: Total VM migration time range estimates (with 95 percent confidence) for VM migration without disk transfer using different ciphers.

Table 6.4 shows the total CPU time consumed by the server and the client during VM migration with disk transfer. Figure 6.2 shows how this time is distributed among different operations (read, write, cryptographic operations). The results are similar to the case without disk transfer. RC4 is more efficient compared to 128-bit AES or 3DES in terms of CPU time. The secure client and server spend a significant amount of time performing cryptographic operations.
Analysis of results

(a) Total migration time with disk transfer (Lenovo T60p as client)

(b) Total migration time with disk transfer (Lenovo T60p as server)

Figure 6.4: Total VM migration time range estimates (with 95 percent confidence) for VM migration with disk transfer using different ciphers.

Figure 6.4 and Figure 6.3 show the total migration time or net migration time for VM migration with disk transfer and without disk transfer respectively. The corresponding data is shown in Table 6.5. The total migration time is directly proportional to the VM RAM size for both the insecure and the secure migration protocol. The insecure protocol is expectedly faster than the secure protocol using any cipher in both contexts. In both cases, the performance of the protocol using 128-bit AES is comparable to RC4. The use of 3DES is considerably more expensive compared to RC4 or 128-bit AES. For some VM RAM sizes, the confidence intervals of the secure protocol using AES and RC4 overlap partially with those of the insecure protocol. We think this is due to lack of sufficient number of profiling samples.
Chapter 6

and we intend to examine it more closely in future work.

It is noted that total migration time with VM disk transfer is smaller when the Lenovo T60p is used as the server than when the Lenovo T60 is used as the server. This maybe due to the Lenovo T60p’s hard disk speed (7200rpm) being higher than the Lenovo T60’s hard disk speed (5400rpm). The protocol versions without disk transfer do not show such a pattern perhaps due to lesser I/O compared to the protocols with disk transfer.

The total migration time is about 8s-13s higher for the secure protocol (using RC4) than for the insecure protocol (for all VM RAM sizes and with and without VM disk transfer). It is likely (from Section 6.2.1) that a large part of this overhead may be due to secure connection establishment. However, since the difference in total migration time between the secure and insecure versions using RC4 is less than 15s irrespective of RAM size, the percentage overhead in total migration time is less for larger sizes of VM RAM. This varies from $\approx 32$ percent (VM RAM size of 128MB) to $\approx 10$ percent (for VM RAM size of 1024MB) for migration without disk transfer. The corresponding percentage overhead range for migration with disk transfer is $\approx 20$ percent (for VM RAM size of 128MB) to $\approx 7$ percent (for VM RAM size of 1024MB).

Although the secure migration protocol consumes more CPU time (for the server and the client), the VM downtime experienced by the end user (which is indicated by the total VM migration time) for a common VM RAM size of 1GB is about 10 percent higher using our secure migration protocol than the insecure migration protocol. We believe this overhead may be tolerable in certain VM migration applications.

6.3 Summary of evaluation results

We compared the performance of our secure migration protocol with its corresponding insecure migration protocol in the context of two VM migration usage scenarios. The first scenario involved transfer of the VM’s disk along with its RAM image. The other scenario transferred just the VM RAM image assuming that the VM disk was on a shared storage accessible to both the client and the server. We measured CPU time and system time for different phases and operations in our protocol. In both scenarios, we found that the secure protocol consumes more CPU time at both the source and the destination and results in higher net migration time than the insecure protocol. RC4 and 128-bit AES are more efficient as underlying ciphers than 3DES. Most of the client and server CPU time is spent in I/O (read and write) in the insecure migration protocol versions while the CPU time of the corresponding secure versions is spent in cryptographic operations (encryption and HMAC) in addition to I/O. The percentage overhead in net migration time perceived by the end user is inversely proportional to the size of the VM RAM for migration. In the case of RC4 (results were similar for 128-bit AES), it varies from $\approx 7\%$ (1024MB VM RAM) and $\approx 20\%$ (128MB VM RAM) for migration with disk transfer and from
### Summary of evaluation results

(a) End-to-end migration time for VM migration without disk transfer

<table>
<thead>
<tr>
<th>VM RAM size (MB)</th>
<th>Insecure</th>
<th>128-bit AES</th>
<th>RC4</th>
<th>3DES</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Server</td>
<td>Client</td>
<td>Server</td>
<td>Client</td>
</tr>
<tr>
<td>128</td>
<td>17.33</td>
<td>16.33</td>
<td>24.33</td>
<td>23.67</td>
</tr>
<tr>
<td>256</td>
<td>25.67</td>
<td>26.67</td>
<td>36.67</td>
<td>35</td>
</tr>
<tr>
<td>512</td>
<td>47.33</td>
<td>46.67</td>
<td>58</td>
<td>57</td>
</tr>
<tr>
<td>768</td>
<td>72.33</td>
<td>73.33</td>
<td>83.67</td>
<td>85</td>
</tr>
<tr>
<td>1024</td>
<td>97.67</td>
<td>100</td>
<td>112.33</td>
<td>111</td>
</tr>
</tbody>
</table>

(b) End-to-end migration time for VM migration with disk transfer

<table>
<thead>
<tr>
<th>VM RAM size (MB)</th>
<th>Insecure</th>
<th>128-bit AES</th>
<th>RC4</th>
<th>3DES</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Server</td>
<td>Client</td>
<td>Server</td>
<td>Client</td>
</tr>
<tr>
<td>128</td>
<td>57.67</td>
<td>51</td>
<td>69.33</td>
<td>63</td>
</tr>
<tr>
<td>256</td>
<td>67.67</td>
<td>61.67</td>
<td>78.33</td>
<td>74.67</td>
</tr>
<tr>
<td>512</td>
<td>89.33</td>
<td>85.33</td>
<td>101.67</td>
<td>99</td>
</tr>
<tr>
<td>768</td>
<td>112.67</td>
<td>110.33</td>
<td>125</td>
<td>122.33</td>
</tr>
<tr>
<td>1024</td>
<td>138.33</td>
<td>136</td>
<td>156.67</td>
<td>151.67</td>
</tr>
</tbody>
</table>

Table 6.5: Average total VM migration time using different ciphers

≃10% (1024MB VM RAM) and ≃32% (128MB VM RAM) for migration without disk transfer. This overhead maybe acceptable in certain scenarios depending upon the hardware and the specific application.
Chapter 7

Conclusion and outlook

Trusted computing technology can be extended for use with virtualization by the use of vTPMs to provide safe storage and ensure system integrity. Among proposed vTPM designs/implementations, currently, software vTPMs are the only viable solution because they neither require virtualization enabled hardware TPMs nor any changes the TPM API. Since vTPMs are used by VM applications and VMs are often migrated between hardware platforms for load balancing and policy enforcement, it is necessary to transfer vTPMs along with their VMs during migration. This requires a vTPM key hierarchy that facilitates migration and a secure VM-vTPM migration protocol.

A vTPM key hierarchy should comply with currently accepted TPM key hierarchy semantics, prevent linking transactions of a vTPM, reduce dependence on TTPs and also facilitate vTPM migration. In our novel key hierarchy, compliance with TPM key hierarchy semantics is achieved by linking the vTPM keys to the host platform. We use the platform hosting the vTPM to generate its credentials to minimize dependence on TTPs. In order to prevent vTPM transaction linking, we recommend use of one-time keys for all communication outside the VM. Finally, our key hierarchy facilitates migration by minimizing vTPM key regeneration or recertification after migration.

Existing solutions to vTPM migration lack a number of security safeguards. We formulated a set of security requirements for a secure VM-vTPM migration protocol in the context of intra-cloud suspended VM migration. Our novel protocol uses mutual source and destination authentication and destination platform attestation for authenticated transfer to a secure platform. Furthermore, it uses encryption, integrity checks and nonces to prevent data leakage, unauthorized modification of VM data and replay attacks respectively. Key establishment using public key certificates ensures source non-repudiation. VM-vTPM association is implicitly maintained by transferring them in a single session. Atomicity of the migration is ensured by preventing duplication of VMs and their vTPMs and enabling recovery from migration failures. Finally, our novel vTPM key hierarchy prevents linking of the vTPM’s transactions.
We implemented our secure protocol and its corresponding insecure version for use in two contexts, on the Xen hypervisor. In the first scenario, the VM’s disk is migrated along with its RAM image to the destination and in the second, the VM’s disk is on a shared storage accessible to both the migration source and destination and only the VM’s RAM is migrated. A performance evaluation of our protocol revealed that the secure protocol consumes more CPU time at both the source and the destination and results in higher net migration time than the insecure protocol. RC4 and 128-bit AES are more efficient as underlying ciphers than 3DES. The percentage overhead in net migration time perceived by the end user is inversely proportional to the size of the VM RAM for migration. In the case of RC4 (results were similar for 128-bit AES), it varies from ≃7% (1024MB VM RAM) and ≃20% (128MB VM RAM) for migration with disk transfer and from ≃10% (1024MB VM RAM) and ≃32% (128MB VM RAM) for migration without disk transfer. This overhead maybe acceptable in certain scenarios depending upon the hardware and the specific application.

Last but not the least, we acknowledge that the hardware used for the implementation and performance evaluation is not representative of real world VM migration scenarios. However, it still allowed the development of a proof-of-concept implementation and a preliminary performance evaluation. In future work, we intend to repeat our profiling using better tools like Intel vTune on more realistic hardware. It would also be useful to understand the overhead of AIK generation (by using a local Privacy-CA) and VM suspension and resumption (using a C implementation). Further, memory usage and network throughput should also be measured in order to understand the exact nature of the overhead imposed by the secure migration protocol. This whole performance evaluation could be repeated for alternate vTPM designs and architectures because this allows identification of viable vTPM architectures.
Bibliography


[10] Tpm main part 1 design principles. [http://www.trustedcomputinggroup.org/resources](http://www.trustedcomputinggroup.org/resources)

[12] Tpm main part 3 commands. [http://www.trustedcomputinggroup.org/resources]


