Providing User Security Guarantees in Public Infrastructure Clouds

Nicolae Paladi, Christian Gehrmann, and Antonis Michalas

Abstract—The infrastructure cloud (IaaS) service model offers improved resource flexibility and availability, where tenants – insulated from the minutiae of hardware maintenance – rent computing resources to deploy and operate complex systems. Large-scale services running on IaaS platforms demonstrate the viability of this model; nevertheless, many organizations operating on sensitive data avoid migrating operations to IaaS platforms due to security concerns. In this paper, we describe a framework for data and operation security in IaaS, consisting of protocols for a trusted launch of virtual machines and domain-based storage protection. We continue with an extensive theoretical analysis with proofs about protocol resistance against attacks in the defined threat model. The protocols allow trust to be established by remotely attesting host platform configuration prior to launching guest virtual machines and ensure confidentiality of data in remote storage, with encryption keys maintained outside of the IaaS domain. Presented experimental results demonstrate the validity and efficiency of the proposed protocols. The framework prototype was implemented on a test bed operating a public electronic health record system, showing that the proposed protocols can be integrated into existing cloud environments.

Index Terms—Security; Cloud Computing; Storage Protection; Trusted Computing

1 Introduction

Cloud computing has progressed from a bold vision to massive deployments in various application domains. However, the complexity of technology underlying cloud computing introduces novel security risks and challenges. Threats and mitigation techniques for the IaaS model have been under intensive scrutiny in recent years [1], [2], [3], [4], while the industry has invested in enhanced security solutions and issued best practice recommendations [5]. From an end-user point of view the security of cloud infrastructure implies unquestionable trust in the cloud provider, in some cases corroborated by reports of external auditors. While providers may offer security enhancements such as protection of data at rest, end-users have limited or no control over such mechanisms. There is a clear need for usable and cost-effective cloud platform security mechanisms suitable for organizations that rely on cloud infrastructure.

One such mechanism is platform integrity verification for compute hosts that support the virtualized cloud infrastructure. Several large cloud vendors have signaled practical implementations of this mechanism, primarily to protect the cloud infrastructure from insider threats and advanced persistent threats. We see two major improvement vectors regarding these implementations. First, details of such proprietary solutions are not disclosed and can thus not be implemented and improved by other cloud platforms. Second, to the best of our knowledge, none of the solutions provides cloud tenants a proof regarding the integrity of compute hosts supporting *their* slice of the cloud infrastructure. To address this, we propose a set of protocols for trusted launch of virtual machines (VM) in IaaS, which provide tenants with a proof that the requested VM instances were launched on a host with an expected software stack.

Another relevant security mechanism is encryption of virtual disk volumes, implemented and enforced at compute host level. While support data encryption at rest is offered by several cloud providers and can be configured by tenants in their VM instances, functionality and migration capabilities of such solutions are severely restricted. In most cases cloud providers maintain and manage the keys necessary for encryption and decryption of data at rest. This further convolutes the already complex data migration procedure between different cloud providers, disadvantaging tenants through a new variation of vendor lock-in. Tenants can choose to encrypt data on the operating system (OS) level within their VM environments and manage the encryption keys themselves. However, this approach suffers from several drawbacks: first, the underlying compute host will still have access encryption keys whenever the VM performs cryptographic operations; second, this shifts towards the tenant the burden of maintaining the encryption software in all their VM instances and increases the attack surface; third, this requires injecting, migrating and later securely withdrawing encryption keys to each of the VM instances with access to the encrypted data, increasing the probability than an attacker eventually obtains the keys. In this paper we present DBSP (domain-based storage protection), a virtual disk encryption mechanism where encryption of data is done directly on the compute host, while the key material necessary for re-generating encryption keys is stored in the volume metadata. This approach allows easy migration of encrypted data volumes and withdraws the control of the cloud provider over disk encryption keys. In addition, DBSP significantly reduces the risk of exposing encryption keys and keeps a low maintenance overhead for the tenant – in the same time providing additional control over the choice of the compute host based on its software stack.

We focus on the Infrastructure-as-a-Service model – in a simplified form, it exposes to its tenants a coherent platform supported by *compute hosts* which operate VM guests that communicate through a virtual network. The system model chosen for this paper is based on requirements identified while migrating a currently deployed, distributed electronic health record (EHR) system to an IaaS platform [6].

1.1 Contribution

We extend previous work applying Trusted Computing to strengthen IaaS security, allowing tenants to place hard security requirements on the infrastructure and maintain exclusive control of the security critical assets. We propose a security framework consisting of three buiding blocks:

- Protocols for trusted launch of VM instances in IaaS;
- Key management and encryption enforcement functions for VMs, providing transparent encryption of persistent data storage in the cloud;
- Key management and security policy enforcement by a Trusted Third Party (TTP);

We describe several contributions that enhance cloud infrastructure with additional security mechanisms:

1. We describe a trusted VM launch (TL) protocol which allows tenants – referred to as *domain managers* – to launch VM instances exclusively on hosts with an attested platform configuration and reliably verify this.
2. We introduce a domain-based storage protection protocol to allow domain managers store encrypted data volumes partitioned according to administrative *domains*.
3. We introduce a list of attacks applicable to IaaS environments and use them to develop protocols with desired security properties, perform their security analysis and prove their resistance against the attacks.
4. We describe the implementation of the proposed protocols on an open-source cloud platform and present extensive experimental results that demonstrate their practicality and efficiency.

1.2 Organization

The rest of this paper is organized as follows. In Section 2 we describe relevant related work on *trusted virtual machine launch* and *cloud storage protection*. In Section 3 we introduce the system model, as well as the threat model and problem statement. In Section 4 we introduce the protocol components, and the TL and DBSP protocols as formal constructions. In Section 5, we provide a security analysis and prove the resistance of the protocols against the defined attacks, while implementation and performance evaluation results are described in Section 6. We discuss the protocol application domain in Section 7 and conclude in Section 8.

2 Related Work

We start with a review of related work on trusted VM launch, followed by storage protection in IaaS.

2.1 Trusted Launch

Santos et al. [1] proposed a "Trusted Cloud Compute Platform" (TCCP) to ensure VMs are running on a trusted hardware and software stack on a remote and initially untrusted host. To enable this, a trusted coordinator stores the list of attested hosts that run a "trusted virtual machine monitor" which can securely run the client's VM. Trusted hosts maintain in memory an individual *trusted key* used for identification each time a client launches a VM. The paper presents a good initial set of ideas for trusted VM launch and migration, in particular the use of a *trusted coordinator*. A limitation of this solution is that the trusted coordinator maintains information about all hosts deployed on the IaaS platform, making it a valuable target to an adversary who attempts to expose the public IaaS provider to privacy attacks.

A decentralized approach to integrity attestation is adopted by Schiffman et al. [2] to address the limited transparency of IaaS platforms and scalability limits imposed by third party integrity attestation mechanisms. The authors describe a trusted architecture where tenants verify the integrity of IaaS hosts through a trusted *cloud verifier* proxy placed in the cloud provider domain. Tenants evaluate the cloud verifier integrity, which in turn attests the hosts. Once the VM image has been verified by the host and countersigned by the cloud verifier, the tenant can allow the launch. The protocol increases the complexity for tenants both by introducing the evaluation of integrity attestation reports of the cloud verifier and host and by adding steps to the trusted VM launch, where the tenant must act based on the data returned from the cloud verifier. Our protocol maintains the VM launch traceability and transparency without relying on a proxy verifier residing in the IaaS. Furthermore, the TL protocol does not require additional tenant interaction to launch the VM on a trusted host, beyond the initial launch arguments.

Platform attestation prior to VM launch is also applied in [7], which introduces two protocols – "TPM-based certification of a Remote Resource" (TCRR) and "VerifyMyVM". With TCRR a tenant can verify the integrity of a remote host and establish a trusted channel for further communication. In "VerifyMyVM", the hypervisor running on an attested host uses an emulated TPM to verify on-demand the integrity of running VMs. Our approach is in many aspects similar to the one in [7] in particular with regard to host attestation prior to VM instance launch. However, the approach in [7] requires the user to *always* encrypt the VM image before instantiation, thus complicating image management. This prevents tenants from using commodity VM images offered by the cloud provider for trusted VM launches. We overcome this limitation and generalize the solution by adding a verification token, created by the tenant and injected on the file system of the VM instance *only* if it is launched on an attested cloud host.

In [8], the authors described a protocol for trusted VM launch on public IaaS using trusted computing techniques. To ensure that the requested VM instance is launched on a host with attested integrity, the tenant encrypts the VM image (along with all injected data) with a symmetric key sealed to a particular configuration of the host reflected in the values of the platform configuration registers (PCR) of the TPM placed on the host. The proposed solution is suitable in trusted VM launch scenarios for enterprise tenants as it requires that the VM image is pre-packaged and encrypted by the client prior to IaaS launch. However, similar to [7], this prevents tenants from using commodity VM images offered by the cloud provider to launch VM instances on trusted cloud hosts. Furthermore, we believe that reducing the number of steps required from the tenant can facilitate the adoption of the trusted IaaS model. We extend some of the ideas proposed in [8], address the above limitations – such as additional actions required from tenants – and also address the requirements towards the launched VM instance and required changes to cloud platforms.

2.2 Secure Storage

Cooper et al. described in in [9] a secure platform architecture based on a secure root of trust for grid environments – precursors of cloud computing. Trusted Computing is used as a method for dynamic trust establishment within the grid, allowing clients to verify that their data wil be protected against malicious host attacks. The authors address the malicious host problem in grid environments, with three main risk factors: trust establishment, code isolation and grid middleware. The solution established a minimal trusted computing base (TCB) by introducing a security manager isolated by the hypervisor from grid services (which are in turn performed within VM instances). The secure architecture is supported by protocols for data integrity protection, confidentiality protection and grid job attestation. In turn, these rely of client attestation of the host running the respective jobs, followed by interaction with the security manager to fulfill the goals of the respective protocols. We follow a similar approach in terms of interacting with a minimal TCB for protocol purposes following host attestation. However, in order to adapt to the cloud computing model we delegate the task of host attestation to an external TTP as well as use TPM functionality to ensure that sensitive cryptographic material can only be accessed on a particular attested host.

In [10], the authors proposed an approach to protect access to outsourced data in an owner-write-users-read case, assuming an "honest but curious service provider". Encryption is done over (abstract) blocks of data, with a different key per block. The authors suggest a key derivation hierarchy based on a public hash function, using the hash function result as the encryption key. The scheme allows to selectively grant data access, uses over-encryption to revoke access rights and supports block deletion, update, insertion and appending. It adopts a lazy revocation model, allowing to indefinitely maintain access to data reachable prior to revocation (regardless of whether it has been accessed before access revocation). While this solution is similar to our model with regard to information blocks and encryption with different symmetric keys, we propose an active revocation model, where the keys are cached for a limited time and cannot be retrieved once the access is revoked.

The "Data-Protection-as-a-Service" (DPaaS) platform [11] balances the requirements for confidentiality and privacy with usability, availability and maintainability. DPaaS focuses on shareable logical data units, confined in isolated partitions (e.g. VMs of language-based features such as Caja, Javascript) or containers, called Secure Execution Environments (SEE). Data units are encrypted with symmetric keys and can be stored on untrusted hardware, while containers communicate through authenticated channels. The authors stress the verifiability of DPaaS using trusted computing and the use of the dynamic root of trust to guarantee that computation is performed on a "secure" platform. The authors posit that DPaaS fulfills confidentiality and privacy requirements and facilitates maintenance, logging and audit; provider migration is one of the aspects highlighted, but not addressed in [11]. Our solution resembles DPaaS in the use of SEE based on software attestation mechanisms offered by the TPM, and in the reliance on full disk encryption to protect data at rest and support for flexible access control management of the data blocks. However, the architecture outlined in [11] does not address bootstrapping the platform (e.g. the VM launch) and provides few details about the key management mechanism for the secure data store. We address the above shortcomings, by describing in detail and evaluating protocols to create and share confidentiality-protected data blocks. We describe cloud storage security mechanisms that allow easy data migration between providers without affecting its confidentiality.

Graf et al. [12] presented an IaaS storage protection scheme addressing access control. The authors analyse access rights management of shared versioned encrypted data on cloud infrastructure for a restricted group and propose a scalable and flexible key management scheme. Access rights are represented as a graph, making a distinction between data encryption keys and encrypted updates on the keys and enabling flexible join/leave client operations, similar to properties presented by the protocols in this paper. Despite its advantages, the requirement for client-side encryption limits the applicability of the scheme in [12] and introduces important functional limitations on indexing and search. In our model, all cryptographic operations are performed on trusted IaaS compute hosts, which are able to allocate more computational resources than client devices.

Santos et al. [13] proposed Excalibur, a system using trusted computing mechanisms to allow decrypting client data exclusively on nodes that satisfy a tenant-specified policy. Excalibur introduces a new trusted computing abstraction, *policy-sealed data* to address the fact that TPM abstractions are designed to protect data and secrets on a standalone machine, at the same time over-exposing the cloud infrastructure by revealing the identity and software fingerprint of individual cloud hosts. The authors extended TCCP [1] to address the limitations of binary-based attestation and data sealing by using property-based attestation [14]. The core of Excalibur is *'the monitor'*, which is a part of the cloud provider, which organises computations across a series of hosts and provides guarantees to tenants. Tenants first decide a policy and receive evidence regarding the status of the monitor along with a public encryption key, and then encrypt their data and policy using ciphertext-policy attribute-based encryption [15]. To decrypt, the stored data hosts receive the decryption key from the monitor who ensures that the corresponding host has a valid status and satisfies the policy specified by the client at encryption time. Our solution is similar to the one in [13], with some important differences: *1)* In contrast with [13] our protocols were implemented as a code extension for Openstack. Furthermore, the presented measurements were made after we deployed the protocols for a part of the Swedish electronic health records management system in an infrastructure cloud. Thus, our measures are considered as realistic since the experiments were done under a real electronic healthcare system; *2)* Excalibur is totally missing a security analysis. Instead authors only present the results of ProVerif (an automated tool) regarding the correctness of their protocol. In addition to that, through our security analysis we introduced a new list of attacks that can be applied to such systems. This is something that is totally missing from related works such as [13] and it can be considered as a great contribution to protocol designers since can avoid common pitfalls and design even better protocols in the future;

In [16] the authors presented a forward-looking design of a cryptographic cloud storage built on an untrusted IaaS infrastructure. The approach aims to provide confidentiality and integrity, while retaining the benefits of cloud storage – availability, reliability, efficient retrieval and data sharing – and ensuring security through cryptographic guarantees rather than administrative controls. The solution requires four client-side components: *data processor, data verifier, credential generator, token generator*. Important building blocks of the solution are: *Symmetric searchable encryption (SSE)*, appropriate in settings where the data consumer is also the one who generates it (efficient for single writer-single reader (SWSR) models); *Asymmetric searchable encryption (ASE)*, appropriate for many writer single reader (MWSR) models, offers weaker security guarantees as the server can mount a dictionary attack against the token and learn the search terms of the client; *Efficient ASE*, appropriate in MWSR scenarios where the search terms are hard to guess, offers efficient search but is vulnerable to dictionary attacks; *Multi-user SSE*, appropriate for single writer/many reader settings, allows the owner to – besides encrypting indexes and generating tokens – revoke user search privileges over data; *Attribute based encryption*, introduced in [17], provides users with a decryption key with certain associated attributes, such that a message can be encrypted using a certain key and a policy. In such a scheme, the message can only be decrypted only if the policy matches the key used to encrypt it; finally, *proofs of storage* allow a client to verify that data integrity has not been violated by the server.

The concepts presented in [16] are promising – especially considering recent progress in searchable encryption schemes [18]. Indeed, integrating searchable and attribute-based encryption mechanisms into secure storage solutions is an important direction in our future work. However, practical application of searchable encryption and attribute-based encryption requires additional research.

Earlier work in [19], [20] described interoperable solutions towards trusted VM launch and storage protection in IaaS. We extend them to create an *integrated* framework that builds a trust chain from the domain manager to the VM instances and data in their administrative domain, and provide additional details, proofs and performance evaluation.

3 System model and Preliminaries

In this section we describe the system and threat model, as well as present the problem statement.

3.1 System Model

We assume an IaaS system model (e.g. OpenStack, a popular open-source cloud platform) as in [21]: providers expose a *quota* of network, computation and storage resources to its tenants – referred to as *domain managers* (Figure 1). Domain managers utilize the quota to launch and operate *VM guests*. Let $\mathcal{DM} = \{DM_1, \ldots, DM_n\}$ be the set of all domain managers in our IaaS. Then, $\mathcal{VM}_i = \{vm_1^i, \ldots, vm_n^i\}$ is the set of all VMs owned by each domain manager DM_i. VM guests operated by $\mathcal{DM}$ are grouped into *domains* (similar to *projects* in OpenStack) which comprise cloud resources corresponding to a particular organization or administrative unit. $\mathcal{DM}$ create, modify, destroy domains and manage access permissions of VMs to data stored in the domains. We refer to $\mathcal{D}_i = \{D_1^i, \ldots, D_n^i\}$ as the set of all domains created by a domain manager DM_i.

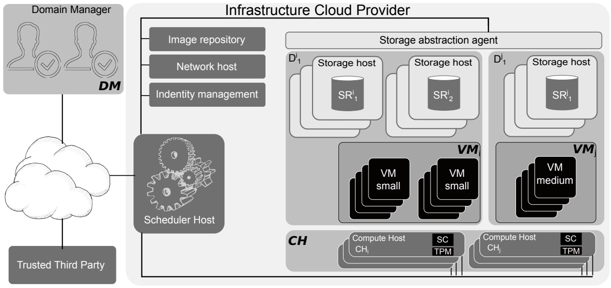


Fig. 1. High level view of the IaaS model introduced in Section 3.

Requests for operations on VMs (launch, migration, termination, etc.) received by the IaaS are managed by a *scheduler* that allocates (reallocates, deallocates) resources from the pool of available *compute hosts* according to a resource management algorithm. We assume in this work compute hosts that are *physical* – rather than virtual – servers. We denote the set of all compute hosts as $\mathcal{CH} = \{CH_1, \ldots, CH_n\}$. We denote a VM instance vm_l^i running on a compute host CH_i by $vm_l^i \mapsto CH_i$ and its unique identifier by $idvm_l^i$.

The *Security Profile* (SP) , defined in [19], is a function of the verified and measured deployment of a *trusted computing base* – a collection of software components measurable during a platform boot. Measurements are maintained in protected storage, usually located on the same platform. We expand this concept in Section 4. Several functionally equivalent configurations may each have a different security profile. We denote the set of all compute hosts that share the same security profile SP_i as $\mathcal{CH}_{SP_i}$. VMs intercommunicate through a virtual network overlay, a "software defined network" (SDN). A domain manager can create arbitrary network topologies in the same domain to interconnect the VMs without affecting network topologies in other domains.

I/O virtualization enables device aggregation and allows to combine several physical devices into a single logical device (with better properties), presented to a VM [22]. Cloud platforms use this to aggregate disparate storage devices into highly available logical devices with arbitrary storage capacity (e.g. *volumes* in OpenStack). VMs are presented with a logical device through a single access interface, while replication, fault-tolerance and storage aggregation are hidden in the lower abstraction layers. We refer to this logical device as *storage resource* (SR); as a storage unit, an SR can be any unit supported by the disk encryption subsystem.

3.2 Threat Model

We share the threat model with [1], [19], [20], [8], which is based on the Dolev-Yao adversarial model [23] and further assumes that privileged access rights can used by a remote adversary $\mathcal{A}DV$ to leak confidential information. $\mathcal{A}DV$, e.g. a corrupted system administrator, can obtain remote access to any host maintained by the IaaS provider, but cannot access the volatile memory of guest VMs residing on the compute hosts of the IaaS provider. This property is based on the closed-box execution environment for guest VMs, as outlined in Terra [24] and further developed in [25], [26].

Hardware Integrity: Media revelations have raised the issue of hardware tampering en route to deployment sites [27], [28]. We assume that the cloud provider has taken necessary technical and non-technical measures to prevent such hardware tampering.

Physical Security: We assume physical security of the data centres where the IaaS is deployed. This assumption holds both when the IaaS provider owns and manages the data center (as in the case of Amazon Web Services, Google Compute Engine, Microsoft Azure, etc.) and when the provider utilizes third party capacity, since physical security can be observed, enforced and verified through known best practices by audit organizations. This assumption is important to build higher-level hardware and software security guarantees for the components of the IaaS.

Low-Level Software Stack: We assume that at installation time, the IaaS provider reliably records integrity measurements of the low-level software stack: the Core Root of Trust for measurement; BIOS and host extensions; host platform configuration; Option ROM code, configuration and data; Initial Platform Loader code and configuration; state transitions and wake events, and a minimal hypervisor. We assume the record is kept on protected storage with read-only access and the adversary cannot tamper with it.

Network Infrastructure: The IaaS provider has physical and administrative control of the network. $\mathcal{ADV}$ is in full control of the network configuration, can overhear, create, replay and destroy all messages communicated between $\mathcal{DM}$ and their resources (VMs, virtual routers, storage abstraction components) and may attempt to gain access to other domains or learn confidential information.

Cryptographic Security: We assume encryption schemes are semantically secure and the $\mathcal{ADV}$ cannot obtain the plain text of encrypted messages. We also assume the signature scheme is unforgeable, i.e. the $\mathcal{ADV}$ cannot forge the signature of DM_i and that the MAC algorithm correctly verifies message integrity and authenticity. We assume that the $\mathcal{ADV}$, with a high probability, cannot predict the output of a pseudorandom function. We explicitly exclude denial-of-service attacks [29] and focus on $\mathcal{ADV}$ that aim to compromise the confidentiality of data in IaaS.

3.3 Problem Statement

The introduced $\mathcal{ADV}$ has far-reaching capabilities to compromise IaaS host integrity and confidentiality. We define a set of attacks available to $\mathcal{ADV}$ in the above threat model.

Given that $\mathcal{ADV}$ has full control over the network communication within the IaaS, one of the available attacks is to inject a malicious program or back door into the VM image, prior to instantiation. Once the VM is launched and starts processing potentially sensitive information, the malicious program can leak data to an arbitrary remote location without the suspicion of the domain manager. In this case, the VM instance will not be a *legitimate* instance and in particular not the instance the domain manager *intended* to launch. We call this type of attack a *VM Substitution Attack*:

Definition 1 (Successful VM Substitution Attack). Assume a domain manager, DM_i, intends to launch a particular virtual machine vm_l^i on an arbitrary compute host in the set $\mathcal{CH}_{SP_i}$. An adversary, $\mathcal{A}DV$, succeeds to perform a **VM substitution attack** if she can find a pair (CH, vm) : $CH \in \mathcal{CH}_{SP_i}$, $vm \in \mathcal{VM}$, $vm \neq vm_l^i$, $vm \mapsto CH$, where vm will be accepted by DM_i as vm_l^i.

A more complex attack involves reading or modifying the information processed by the VM directly, from the logs and data stored on CH or from the representation of the guest VMs' drives on the CH file system. This might be non-trivial or impossible with strong security mechanisms deployed on the host; however, $\mathcal{A}DV$ may attempt to circumvent this through a so-called *CH Substitution Attack* – by launching the guest VM on a compromised CH.

Definition 2 (Successful CH Substitution Attack). Assume DM_i wishes to launch a VM vm_l^i on a compute host in the set $\mathcal{CH}_{SP_i}$. An adversary, $\mathcal{A}DV$, succeeds with a **CH substitution attack** *iff* $\exists\ vm_l^i \mapsto CH_j$, $CH_j \in \mathcal{CH}_{SP_j}$, $SP_j \neq SP_i$: vm_l^i will be accepted by DM_i.

Depending on the technical expertise of DM_i, $\mathcal{ADV}$ may still take the risk of deploying a concealed – but feature-rich – malicious program in the guest VM and leave a fall back option in case the malicious program is removed or prevented from functioning as intended. $\mathcal{ADV}$ may choose a *combined VM and CH substitution attack*, which allows a modified VM to be launched on a compromised host and present it to DM_i as the intended VM:

Definition 3 (Successful Combined VM and CH Substitution Attack). Assume a domain manager, DM_i, wishes to launch a virtual machine vm_l^i on a compute host in the set $\mathcal{CH}_{SP_i}$. An adversary, $\mathcal{A}DV$, succeeds to perform a **combined CH and VM substitution attack** if she can find a pair (CH, vm), $CH \in \mathcal{CH}_{SP_j}$, $SP_j \neq SP_i$, $vm \in \mathcal{VM}$, $vm \neq vm_l^i, vm \mapsto CH$, where vm will be accepted by DM_i as vm_l^i.

Denote by $\mathcal{D}_{vm}^i$ the set of storage domains that $vm \in \mathcal{VM}$, $vm \mapsto CH_i$ can access. We define a *successful storage compute host substitution attack* as follows[1]:

Definition 4 (Successful Storage CH Substitution Attack). A DM_i wishes to launch or has launched an arbitrary virtual machine vm_l^i on a compute host in the set $\mathcal{CH}_{SP_i}$. An adversary $\mathcal{A}DV$ succeeds with a **storage CH substitution attack** if she manages to launch $vm_l^i \mapsto CH_j$, $CH_j \in \mathcal{CH}_{SP_j}$, $SP_j \neq SP_i$ and $\mathcal{D}_{vm_l^i}^i \cap \mathcal{D}_{vm_l^i}^j \neq \emptyset$.

If access to the data storage resource is given to all VMs launched by DM_i, $\mathcal{A}DV$ may attempt to gain access by

[1] In this definition we exclude the possibility of legal domain sharing which would be a natural requirement for most systems. However, with our suggested definition, the legal sharing case can be covered by extending the domain manager role such that it is allowed not to a distinct entity but a role that is possibly shared between domain managers that belong to different organizations.

launching a VM that *appears* to have been launched by DM_i. Then, $\mathcal{ADV}$ would be able to leak data from the domain owned by DM_i to other domains. This infrastructure-level attack would not be detected by DM_i and requires careful consideration. A formal definition of the attack[1] follows.

Definition 5 (Successful Domain Violation Attack). Assume DM_i has created the domains in the set $\mathcal{D}_i$. An adversary $\mathcal{ADV}$ succeeds to perform a **domain violation attack** if she manages to launch an arbitrary VM, vm_m^j on an arbitrary host CH_j, i.e. $vm_m^j \mapsto CH_j$, where $\mathcal{D}^j_{vm_m^j} \cap \mathcal{D}_i \neq \emptyset$.

4 Protocol Description

We now describe two protocols that constitute the core of this paper's contribution. These protocols are successively applied to deploy a cloud infrastructure providing additional user guarantees of cloud host integrity and storage security. For protocol purposes, each domain manager, secure component and trusted third party has a public/private key pair (pk/sk). The private key is kept secret, while the public key is shared with the community. We assume that during the initialization phase, each entity obtains a certificate via a trusted certification authority. We first describe the cryptographic primitives used in the proposed protocols, followed by definitions of the main protocol components.

4.1 Cryptographic Primitives

The set of all binary strings of length n is denoted by $\{0,1\}^n$, and the set of all finite binary strings as $\{0,1\}^*$. Given a set U, we refer to the i^{th} element as v_i. Additionally, we use the following notations for cryptographic operations throughout the paper:

- For an arbitrary message $m \in \{0,1\}^*$, we denote by $c = \mathsf{Enc}\,(K, m)$ a symmetric encryption of m using the secret key $K \in \{0,1\}^*$. The corresponding symmetric decryption operation is denoted by $m = \mathsf{Dec}(K, c) = \mathsf{Dec}(K, \mathsf{Enc}(K, m))$.
- We denote by pk/sk a public/private key pair for a public key encryption scheme. Encryption of message m under the public key pk is denoted by $c = \mathsf{Enc}_{\mathsf{pk}}\,(m)$[2] and the corresponding decryption operation by $m = \mathsf{Dec}_{\mathsf{sk}}(c) = \mathsf{Dec}_{\mathsf{sk}}(\mathsf{Enc}_{\mathsf{pk}}(m))$.
- A digital signature over a message m is denoted by $\sigma = \mathsf{Sign}_{\mathsf{sk}}(m)$. The corresponding verification operation for a digital signature is denoted by $b = \mathsf{Verify}_{\mathsf{pk}}(m, \sigma)$, where $b = 1$ if the signature is valid and $b = 0$ otherwise.
- A Message Authentication Code (MAC) using a secret key K over a message m is denoted by $\mu = \mathsf{MAC}(K, m)$.
- We denote by $\tau = \mathsf{RAND}(n)$ a random binary sequence of length n, where $\mathsf{RAND}(n)$ represents a random function that takes a binary length argument n as input and gives a random binary sequence of this length in return[3].

4.2 Protocol Components

Disk encryption subsystem: a software or hardware component for data I/O encryption on storage devices, capable to encrypt storage units such as hard drives, software RAID volumes, partitions, files, etc. We assume a software-based subsystem, such as *dm-crypt*, a disk encryption subsystem using the Linux kernel Crypto API.

Trusted Platform Module (TPM): a hardware cryptographic co-processor following specifications of the Trusted Computing Group (TCG) [30]; we assume $\mathcal{CH}$ are equipped with a TPM v1.2. The tamper-evident property facilitates monitoring $\mathcal{CH}$ integrity and strengthens the assumption of physical security. An active TPM records the platform boot time software state and stores it as a list of hashes in platform configuration registers (PCRs). TPM v1.2 has 16 PCRs reserved for *static measurements* (PCR0 - PCR15), cleared upon a hard reboot. Additional runtime resettable registers (PCR16-PCR23) are available for *dynamic measurements*. *Endorsement keys* are an asymmetric key pair stored inside the TPM in the trusted platform supply chain, used to create an *endorsement credential* signed by the TPM vendor to certify the TPM specification compliance. A message encrypted ("bound") using a TPM's public key is decryptable only with the private key of the same TPM. *Sealing* is a special case of binding – bound messages are only decryptable in the platform state defined by PCR values. *Platform attestation* allows a remote party to authenticate a target platform and obtain a guarantee that it – up to a certain level in the boot chain – runs software that is identical to the expected one. To do this, an attester requests – accompanied by a nonce – the target platform to produce an attestation quote and the measurement aggregate, or Integrity Measurement List (IML). The TPM generates the attestation quote – a signed structure that includes the IML and the received nonce – and returns the quote and the IML itself. The attestation quote is signed with the TPMs *Attestation Identity Key (AIK)*. The exact IML contents are implementation-specific, but should contain enough data to allow the *verifier* to establish the target platform [31] integrity. We refer to [30] for a description of the TPM, and to [7], [19], [20] for protocols that use TPM functionality.

Trusted Third Party (TTP): an entity trusted by the other components. TTP verifies the TPM endorsement credentials on hosts operated by the cloud provider and *enrolls* the respective TPMs' AIKs by issuing a signed AIK certificate. We assume that TTP has access to an access control list (ACL) describing access and ownership relations between $\mathcal{DM}$ and $\mathcal{D}$. Furtermore, TTP communicates with $\mathcal{CH}$ to exchange integrity attestation data, authentication tokens and cryptographic keys. TTP can attest *platform integrity* based on the integrity attestation quotes and the valid AIK certificate from a TPM, and seal data to a trusted host configuration. Finally, TTP can verify the authenticity of $\mathcal{DM}$ and perform necessary cryptographic operations. In this paper, we treat the TTP as a "black box" with a limited, well-defined functionality, and omit its internals. Availability of the TTP is essential in the cloud scenario – we refer the reader to the rich body of work on fault tolerance for approaches to building highly available systems.

[2] Alternative notations used for clarity are $\{m\}_{\mathsf{pk}}$ or $\langle m \rangle_{\mathsf{pk}}$.

[3] We assume that a true random function in our constructions is replaced by a pseudorandom function the input-output behaviour of which is "computationally indistinguishable" from that of a true random function.

Secure Component (SC): this is a verifiable execution module performing confidentiality and integrity protection operations on VM guest data. SC is present on all $\mathcal{CH}$ and is responsible for enforcing the protocol; it acts as a mediator between the $\mathcal{DM}$ and the TTP and forwards the requests from $\mathcal{DM}$ to either the TTP or the disk encryption subsystem. SC must be placed in an isolated execution environment, as in the approaches presented in [25], [26].

4.3 Trusted Launch Construction

We now present our construction for the TL, with four participating entities: *domain manager*, *secure component*, *trusted third party* and *cloud provider* (with the 'scheduler' as part of it). TL comprises a public-key encryption scheme, a signature scheme and a token generator. Figure 2 shows the protocol message flow (some details omitted for clarity).

TL.Setup : Each entity obtains a public/private key pair and publishes its public key. Below we provide the list of key pairs used in the following protocol:

- $(\mathsf{pk}_{\mathsf{DM_i}}, \mathsf{sk}_{\mathsf{DM_i}})$ – public/private key pair for DM_i;
- $(\mathsf{pk}_{\mathsf{TTP}}, \mathsf{sk}_{\mathsf{TTP}})$ – public/private key pair for TTP;
- $(\mathsf{pk}_{\mathsf{TPM}}, \mathsf{sk}_{\mathsf{TPM}})$ – TPM bind key pair;
- $(\mathsf{pk}_{\mathsf{AIK}}, \mathsf{sk}_{\mathsf{AIK}})$ – TPM attestation identity key pair;

TL.Token : To launch a new VM instance vm_l^i, DM_i generates a token by executing $\tau = \mathsf{RAND}(n)$ and calculates the hash (H_1) of the VM image (vm_l^i) intended for launch, the hash (H_2) of $\mathsf{pk}_{\mathsf{DM_i}}$, and the required security profile SP_i. Finally, $\mathcal{D}^i_{vm_l^i}$ describes the set of domains that vm_l^i with the identifier $idvm_l^i$ shall have access to; the six elements are concatenated into: $m_1 = \left\{\tau \,\|H_1 \,\|H_2 \,\|SP_i\|idvm_l^i\|\mathcal{D}^i_{vm_l^i}\right\}$. DM_i encrypts m_1 with $\mathsf{pk}_{\mathsf{TTP}}$ by running $c_1 = \mathsf{Enc}_{\mathsf{pk}_{\mathsf{TTP}}}(m_1)$.

Next, DM_i generates a random nonce r and sends the following arguments to initiate a trusted VM launch procedure: $\langle c_1, SP_i, \mathsf{pk}_{\mathsf{DM_i}}, r\rangle$, where c_1 is the encrypted message generated in TL.Token, SP_i is the requested security profile and $\mathsf{pk}_{\mathsf{DM_i}}$ is the public key of DM_i. The message is signed with $\mathsf{sk}_{\mathsf{DM_i}}$, producing σ_{DM_i}. Upon reception, the scheduler assigns the VM launch to an appropriate host with a security profile SP_i, e.g. host CH_i. In all further steps, the nonce r and the signature of the message are used to verify the freshness of the received messages.

Upon reception, SC verifies message integrity and TL.Token freshness by checking respectively the signature σ_{DM_i} and nonce r. When SC first receives a TL.Request message, it uses the local TPM to generate a new pair of TPM-based public/private bind keys, $(\mathsf{pk}_{\mathsf{TPM}}, \mathsf{sk}_{\mathsf{TPM}})$, which can be reused for future launch requests, to avoid the costly key generation procedure. Keys can be periodically regenerated according to a cloud provider-defined policy. To prove that the bind keys are non-migratable, PCR-locked, public/private TPM keys, SC retrieves the `TPM_CERTIFY_INFO` structure, signed with the TPM attestation identity key $\mathsf{pk}_{\mathsf{AIK}}$ [30] using the `TPM_CERTIFY_KEY` TPM command; we denote this signed structure by $\sigma_{\mathtt{TCI}}$. `TPM_CERTIFY_INFO` contains the bind key hash and the PCR value required to use the key; PCR values must not necessarily be in a trusted state to create a trusted bind key pair. This mechanism is explained in further detail in [19].

Next, SC sends an attestation request (TL.AttestRequest) to the TTP, containing the encrypted message (c_1) generated by DM_i in TL.Token, the nonce r and the attestation data (*AttestData*), used by the TTP to evaluate the security profile of CH_i and generate the corresponding TPM bind keys. SC also requests the TPM to sign the message with $\mathsf{sk}_{\mathsf{AIK}}$, producing σ_{AIK}. *AttestData* includes the following:

- the public TPM bind key $\mathsf{pk}_{\mathsf{TPM}}$;
- the `TPM_CERTIFY_INFO` structure;
- σ_{TCI}: signature of`TPM_CERTIFY_INFO` using $\mathsf{sk}_{\mathsf{AIK}}$;
- IML, the integrity measurement list;
- the TCI-certificate;

Upon reception, TTP verifies the integrity and freshness of TL.AttestRequest, checking respectively the signature σ_{AIK} and nonce r. Next, TTP verifies – according to its ACL – the set $\mathcal{D}^i_{vm_l^i}$ to ensure that DM_i is authorised to allow access to the requested domains for vm_l^i and decrypts the message $m_1 := \mathsf{Dec}_{\mathsf{sk}_{\mathsf{TTP}}}(c_1)$, decomposing it into τ, H_1, H_2, SP_i. Finally, TTP runs an attestation scheme to validate the received attestation information and generate a new attestation token.

Definition 6 (Attestation Scheme). An attestation scheme, denoted by TL.Attestation, is defined by two algorithms (AttestVerify, AttestToken) such that:

1. AttestVerify is a deterministic algorithm that takes as input the encrypted message from the requesting DM_i and attestation data, $\langle c_1, AttestData\rangle$, and outputs a result bit b. If the attestation result is positive, $b = 1$; otherwise, $b = 0$. We denote this by $b := \mathsf{AttestVerify}(c_1, \sigma_{AIK}, AttestData)$.
2. AttestToken is a probabilistic algorithm that produces a TPM-sealed attestation token. The input of the algorithm is the result of AttestVerify, the message m to be sealed and the CH $AttestData$. If AttestVerify evaluates to $b = 1$, the algorithm outputs an encrypted message c_2. We write this as $c_2 \leftarrow \mathsf{AttestToken}(b, m, AttestData)$. Otherwise, if AttestVerify evaluates to $b = 0$, AttestToken returns $\perp$.

In the attestation step, TTP first runs AttestVerify to determine the trustworthiness of the target CH_i. In AttestVerify, TTP verifies the signature σ_{TCI} and σ_{AIK} against a valid AIK certificate contained in $AttestData$ and examines the entries provided in the IML. AttestVerify returns $b = 0$ and TTP exits the protocol if the entries differ from values expected for the security profile SP_i. Otherwise, AttestVerify returns $b = 1$ and TTP runs AttestToken to generate a new encrypted attestation token for CH_i. Having verified that the entries in IML conform to the security profile SP_i, TTP generates a symmetric domain encryption key, DK_i, to protect the communication between the SC and TTP in future exchanges. Finally, TTP seals $m_2 = \left\{\tau\|H_1\|H_2\|DK_i\|idvm_l^i\right\}$ to the trusted platform configuration of CH_i, using the key $\mathsf{pk}_{\mathsf{TPM}}$ received through the attestation request. The encrypted message $(c_2 \leftarrow \mathsf{AttestToken}(b, m_2, AttestData),\ r)$, along with a signature (σ_{TTP}) produced using $\mathsf{sk}_{\mathsf{TTP}}$ is returned to SC.

Upon reception, SC checks the message integrity and freshness before unsealing it using the corresponding TPM bind key $\mathsf{sk}_{\mathsf{TPM}}$. The encrypted message is unsealed to the

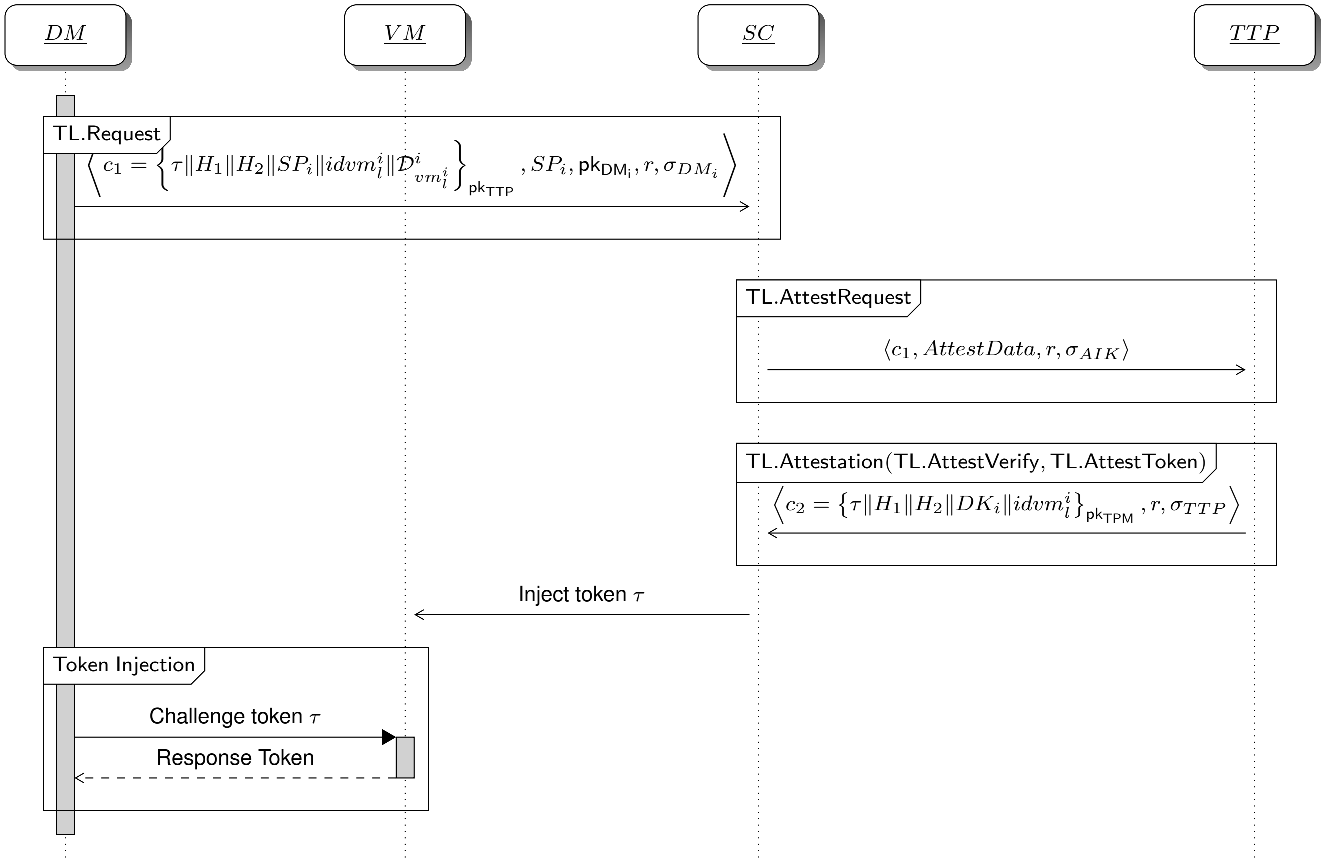

DM
VM
SC
TTP
TL.Request
$\left\langle c_1 = \left\{\tau\|H_1\|H_2\|SP_i\|idvm_l^i\|\mathcal{D}^i_{vm_l^i}\right\}_{\mathsf{pk_{TTP}}}, SP_i, \mathsf{pk_{DM_i}}, r, \sigma_{DM_i}\right\rangle$
TL.AttestRequest
$\langle c_1, AttestData, r, \sigma_{AIK}\rangle$
TL.Attestation(TL.AttestVerify, TL.AttestToken)
$\left\langle c_2 = \{\tau\|H_1\|H_2\|DK_i\|idvm_l^i\}_{\mathsf{pk_{TPM}}}, r, \sigma_{TTP}\right\rangle$
Inject token τ
Token Injection
Challenge token τ
Response Token

Fig. 2. Message Flow in the Trusted VM Launch Protocol.

plain text $m_2 = \{\tau\|H_1\|H_2\|DK_i\|idvm_l^i\}$ only if the platform state of CH_i has remained unchanged. SC calculates the hash (H_1') of the VM image supplied for launch and verifies that its identifier matches the expected identifier $idvm_l^i$; SC also calculates the hash of $\mathsf{pk}_{\mathsf{DM_i}}$ received from the cloud provider, denoted by H_2'. Finally, SC verifies that $H_1 = H_1'$ and only in that case injects τ into the VM image. Likewise, SC verifies that the public key registered by DM_i with the cloud provider in step TL.Setup has not been altered, i.e. $H_2 = H_2'$ and only in that case injects $\mathsf{pk}_{\mathsf{DM_i}}$ into the VM image prior to launching it.

In the last protocol step, DM_i verifies that vm_l^i has been launched on a trusted platform with security profile SP_i, while vm_l^i verifies the authenticity of DM_i. This is done by establishing a secure pre-shared key TLS session [32] between vm_l^i and DM_i using τ as the pre-shared secret.

4.4 Domain-Based Storage Protection Construction

We now continue with a description of the DBSP protocol.

Along with three of the entities already active in the TL protocol – *domain manager*, *secure component*, the *trusted third party* – DBSP employs a fourth one: *the storage resource*. In this case, DM_i interacts with the other protocol components through a VM instance vm_l^i running on CH_i. We assume that vm_l^i has been launched following the TL protocol. The DBSP protocol includes a public and a private encryption scheme, a pseudorandom function for domain key generation, a signature scheme and a random generator. Figure 3 presents the DBSP protocol mesage flow.

DBSP.Setup: We assume that in TL.Setup, each entity has obtained a public/private key pair and published pk.

Assume DM_i requests access for a certain VM vm_l^i to a storage resource SR_i in the domain $D_k^i \in \mathcal{D}^i_{vm_l^i}$. The request is intercepted by the SC, which proceeds to retrieve from TTP a symmetric encryption key for the domain D_k^i.

DBSP.DomKeyReq: SC sends to TTP a request to generate keys for the domain D_k^i. The request contains the target storage resource SR_i, hash H_2 of $\mathsf{pk}_{\mathsf{DM_i}}$, the nonce r and $meta_k^i$, containing the unique domain identifier and the security profile required to access the domain D_k^i, i.e., $meta_k^i = \{D_k^i, SP_i\}_{\mathsf{pk_{TTP}}}$; SC uses the symmetric key DK_i received during TL.Attestation to protect message confidentiality, and the local TPM to sign the message with $\mathsf{sk_{AIK}}$, producing σ_{AIK} (see DBSP.DomKeyReq in Figure 3).

Upon the reception of DBSP.DomKeyReq, TTP verifies the freshness and integrity of the request and proceeds to the next protocol step, DBSP.DomKeyGen, only if this verification succeeds.

DBSP.DomKeyGen: A probabilistic algorithm enabling TTP to generate a symmetric encryption key (K_k^i) and integrity key (IK_k^i) for a domain D_k^i. TTP generates a nonce using a random message $m_i \in \{0,1\}^n$ by executing $n_i = \mathsf{RAND}\,(m_i)$. Next, TTP uses a PRF to generate the keys for domain D_k^i, by evaluating the following:

$$K_k^i = PRF\left(K_{TTP}, D_k^i\|SP_i\|n_i\right),$$
$$IK_k^i = PRF\left(K_{TTP}, D_k^i\|n_i\right),$$

where K_{TTP} is a master key that does not leave the security

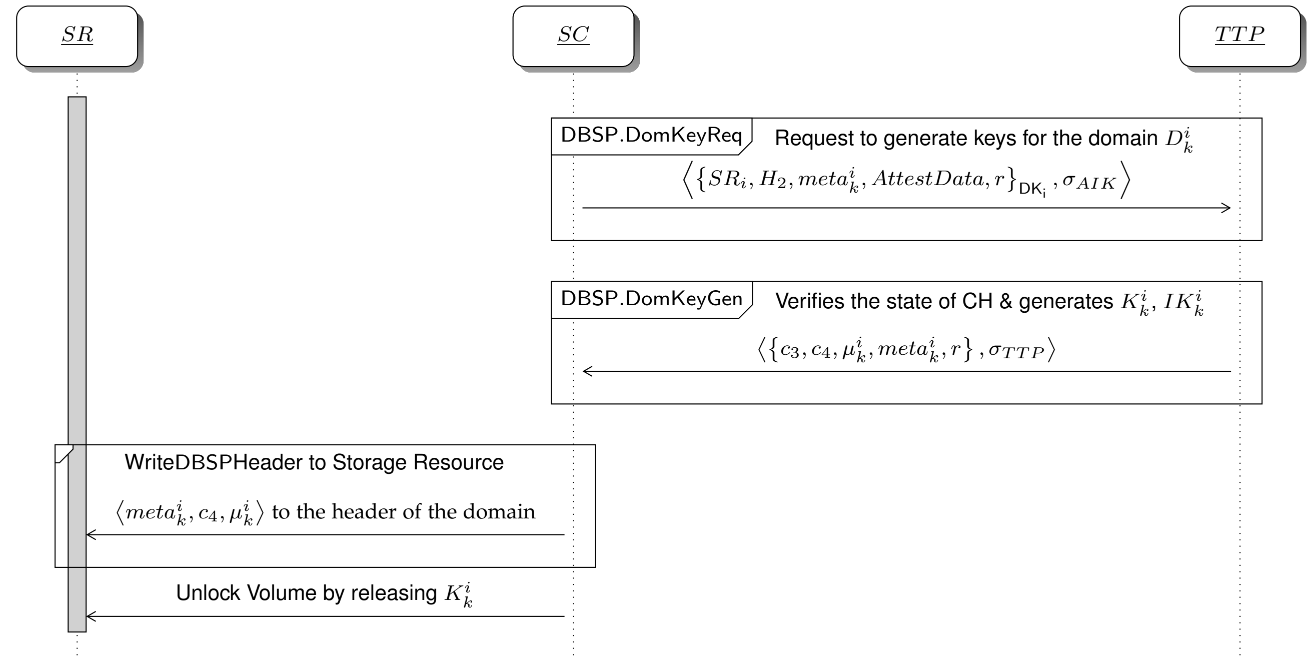

SR
SC
TTP
DBSP.DomKeyReq
Request to generate keys for the domain D_k^i
$\left\langle \{SR_i, H_2, meta_k^i, AttestData, r\}_{\mathsf{DK_i}}, \sigma_{AIK} \right\rangle$
DBSP.DomKeyGen
Verifies the state of CH & generates K_k^i, IK_k^i
$\langle \{c_3, c_4, \mu_k^i, meta_k^i, r\}, \sigma_{TTP} \rangle$
WriteDBSPHeader to Storage Resource
$\langle meta_k^i, c_4, \mu_k^i \rangle$ to the header of the domain
Unlock Volume by releasing K_k^i

Fig. 3. Message Flow in the Domain-Based Storage Protection Protocol.

perimeter of TTP, K_k^i is a symmetric encryption key to confidentiality protect the data and IK_k^i a symmetric key to verify the integrity of the stored data.

TTP seals K_k^i and IK_k^i to the trusted configuration of CH_i by calculating $c_3 = \mathsf{Enc}_{\mathsf{pk}_{\mathsf{TPM}}}\left(K_k^i \| IK_k^i\right)$. TTP encrypts the generated nonce n_i and the provided security profile SP_i by evaluating $c_4 = \mathsf{Enc}_{\mathsf{K}_{\mathsf{TTP}}}\left(n_i \| SP_i\right)$ to later use it for verification. Next, TTP generates a message authentication code μ by evaluating $\mu_k^i = \mathsf{MAC}(K_{TTP}, n_i \| SP_i)$. The domain key generation algorithm is denoted by $(c_3, c_4, \mu_k^i) \leftarrow \mathsf{DBSP.DomKeyGen}(n_i, K_{TTP}, \mathsf{sk}_{\mathsf{TPM}})$.

Having generated the domain key, TTP responds to the DBSP.DomKeyReq by sending $\{c_3, c_4, \mu_k^i, meta_k^i, r\}$ with the signature σ_{TTP}. Upon reception, SC first verifies message integrity and freshness, and calls the local TPM to unseal c_3, producing $K_k^i \| IK_k^i$ if and only if CH_i remains in the earlier trusted state. Next, SC stores $meta_k^i$, c_4 and μ_k^i in the domain header and uses K_k^i, IK_k^i as inputs to the disk encryption subsystem on CH_i, which decrypts and verifies the data integrity of the mounted volume hosting D_k^i before providing plain text access to vm_l^i.

To recreate the encryption and integrity keys for the domain D_k^i, SC sends a request similar to DBSP.DomKeyReq, adding to the message the values c_4 and μ_k^i, which are stored in the domain header. Upon reception, TTP verifies the integrity of the received value c_4 by calculating $\mu_k^i = \mathsf{MAC}(K_{TTP}, n_i \| SP_i)$. If the integrity verification of c_4 is positive, TTP decrypts it to $n_i \| SP_i = \mathsf{Dec}_{\mathsf{sk}_{\mathsf{TTP}}}\left(c_4\right)$ and calculates the domain key as in DBSP.DomKeyGen, using the existing token n_i instead of generating a new one[4].

5 Security Analysis

We now analyse the TL and DBSP protocols in the presence of an adversary. We prove the security of both schemes through a theoretical analysis, showing that our protocols are resistant to the attacks presented in Section 3.3.

Proposition 1 (VM Substitution Soundness). The TL protocol is sound against the VM substitution attack.

Proof: An adversary $\mathcal{ADV}$ trying to launch $vm \neq vm_l^i$ on CH can only get vm accepted by DM_i if the last mutual authentication step in the trusted launch procedure is successful. In turn, this step only succeeds if at least one of the following two options is true:

a. The secure component SC uses a different token, $\tau' \neq \tau$ accepted by DM_i in the final secure channel establishment.
b. The secure component SC on CH uses the very same token τ used by DM_i when launching vm_l^i.

Option a can only succeed if $\mathcal{ADV}$ can break the mutual authentication in the secure channel setting. Given that the selected secure channel scheme is sound and τ is sufficiently long and selected using a sound random generation process, the $\mathcal{ADV}$ fails to break the last protocol step. Hence, as long as the secure channel protocol is sound, the overall protocol construction is also sound against this attack option.

Option b can only succeed if the adversary either manages to guess a value $\tau' = \tau$ when launching vm or manages to either obtain τ when DM_i launches vm_l^i or replace the association between τ and vm_l^i with an association between τ and vm when DM_i launches vm_l^i, by attacking any of the protocol steps preceding the final mutual authentication step. A successful attack in this case has the probability $\tau' = \tau$ equals to $1/2^n$, where n is the length of the token value and is infeasible if n is large enough. Below, we show why the adversary also fails with respect to the last option.

- TL.Token. Assume the adversary intercepts the TL.Token message. Then the adversary has two options: she might either try to modify the TL.Token

[4] Key retrieval is currently not covered in the security analysis due to space limitations

message (option 1) with the goal to replace the association between τ and the vm_l^i with τ and vm, or she might try obtain the secret value τ (option 2) and then launch vm with this τ value on an arbitrary valid provider platform. We discuss both these options below.

- $\mathsf{TL.Token}$ *Option 1*: A modification can only be achieved by the adversary by either breaking the public key encryption scheme used to produce c_1 or trying to make this modification on c_1 by direct modification (without first decrypting it) and sign the modified c_1 with an own selected private key. The former option fails due to the assumption of public key encryption scheme soundness and the latter due to that modifying a public encrypted structure without knowledge of the private key is infeasible.
- $\mathsf{TL.Token}$ *Option 2*: Direct decryption of c_1 fails due to the assumption of soundness of the public key encryption scheme used to produce c_1. The only remaining alternative for the adversary is relaying the $\mathsf{TL.Token}$ to a platform $CH' \in \mathcal{CH}_{SP_i}$, which is under the full control of the adversary. Further, $\mathcal{ADV}$ follows the protocol and issues the command $\mathsf{TL.AttestRequest}$ using the intercepted c_1, *AttestData* and σ_{AIK}. However, this fails at the $\mathsf{TL.Attestation}$ step since $AttestData$ does not contain a valid AIK certificate unless the adversary has managed to get control of a valid platform in the provider network with a valid certificate *or* she has managed to break the AIK certification scheme. The former option violates the assumption of physical security of the provider computing resources while the latter option violates the assumption of a sound public key and AIK certification schemes.

- $\mathsf{TL.AttestRequest}$. The adversary could either try to impersonate this message with the goal of obtaining τ or the association between τ and vm_l^i. This impersonation attempt fails as the whole sent structure is signed with the $\mathsf{pk_{AIK}}$ with a secure public key signing scheme. Furthermore, attempts to resend an old *valid* $\mathsf{TL.AtttestRequest}$ fail as the H_1 verification that the SC receives in return fails as it does point on the *old* VM. Similarly, any attempts to modify $\mathsf{TL.AttestRequest}$ fail as the whole structure is signed with a secure signature scheme.
- $\mathsf{TL.Attestation}$. Any attempt by the adversary to obtain τ would be equal to breaking the public key encryption of $\mathsf{TL.AttestToken}$. Similarly, any attempt to modify c_2 fails due to the fact that modification of a public encrypted structure without knowledge of the private key is unfeasible if the public key encryption scheme is sound. Any attempt by the adversary to replace an old recorded valid $\mathsf{TL.AttestToken}$ message fails as such messages do contain a VM image hash H_1 different than the one expected by the SC. □

Proposition 2 (CH Substitution Soundness). The TL protocol is sound against the CH substitution attack.

Proof: DM_i intends to launch a virtual machine vm_l^i on an arbitrary compute host CH_i with a security profile SP_i. An adversary $\mathcal{ADV}$ trying to launch vm_l^i on $CH_j \in \mathcal{CH}_{SP_j}$, $SP_j \neq SP_i$, can only get vm_l^i accepted by DM_i if the last mutual authentication step in the trusted launch procedure is successful. In turn, this step can only succeed if at least one of the following two options is true:

a. The secure component SC is using a different token, $\tau' \neq \tau$ that is accepted by DM_i in the final secure channel establishment.
b. The secure component SC on CH_j is using the very same token τ used by DM_i when launching vm_l^i.

Option a is impossible as proved in Proposition 1. *Option b* can only succeed if the adversary either manages to guess a value $\tau' = \tau$ when launching vm_l^i or manages to induce the TTP to seal the token τ to the configuration of CH_j. Finding $\tau' = \tau$ is infeasible for the adversary as shown in Proposition 1. Below, we show why the adversary also fails with respect to the second option.

Assume $\mathcal{ADV}$ intercepts the $\mathsf{TL.Token}$ message. Then it has two options: either attempt to launch vm_l^i on a compute host $CH_j \notin \mathcal{CH}_{SP_i}$ or on $CH_j \in \mathcal{CH}_{SP_i}$.

- $\mathsf{TL.Token}$ $CH_j \notin \mathcal{CH}_{SP_i}$: The $\mathcal{ADV}$ can replace the following information from the $\mathsf{TL.Token}$ message: SP_i with SP_j, $\mathsf{pk_{DM_i}}$ with $\mathsf{pk}_{\mathcal{ADV}}$, which is a public key generated by the $\mathcal{ADV}$ and σ_{c_1} with $\sigma_{\mathcal{ADV}} = \mathsf{Sign}_{\mathsf{sk}_{\mathcal{ADV}}}(\mathsf{c_1})$. By doing this, she can successfully proceed beyond the $\mathsf{TL.AttestRequest}$ step since SC is not able to detect the substitution. However, this attack fails at the $\mathsf{TL.Attestation}$ step since the $AttestData$ sent to the TTP evaluates to a security profile $SP_j \neq SP_i$ in contradiction with the preference of DM_i contained in c_1.
- $\mathsf{TL.Token}$ $CH_j \in \mathcal{CH}_{SP_i}$: The $\mathcal{ADV}$ can replace the following information from the $\mathsf{TL.Token}$ message: $\mathsf{pk_{DM_i}}$ with $\mathsf{pk}_{\mathcal{ADV}}$, which is a public key generated by the $\mathcal{ADV}$ and σ_{c_1} with $\sigma_{\mathcal{ADV}} = \mathsf{Sign}_{\mathsf{sk}_{\mathcal{ADV}}}(\mathsf{c_1})$. By doing this, he can successfully proceed beyond the $\mathsf{TL.AttestRequest}$ step since SC is unable to detect the substitution. However, this attack fails at the $\mathsf{TL.Attestation}$ step since the $\mathsf{pk_{AIK}}$ key used to produce the signature σ_{AIK} is not among the keys enrolled with the TTP according to Section 4.2.

The cases of $\mathsf{TL.AttestRequest}$ and $\mathsf{TL.Attestation}$ fail as demonstrated in Proposition 1. □

Proposition 3 (Combined VM and CH Substitution Soundness). The TL protocol is sound against the VM and CH substitution attack.

Proof: The exculpability of the VM substitution attack and the CH substitution attack implies that the TL protocol is secure against the combined VM and CH substitution attack. □

Proposition 4 (Storage CH Substitution Soundness). The DBSP protocol is sound against the storage CH attack.

Proof: Adversary $\mathcal{ADV}$ can only succeed with a storage CH substitution attack if she manages to launch a VM instance

$vm_l^i \mapsto CH_i$, $CH_i \in \mathcal{CH}_{SP_i}$ on a host $CH_j \in \mathcal{CH}_{SP_j}$, $SP_j \neq SP_i$ and $\mathcal{D}^i_{vm_l^i} \cap \mathcal{D}^j_{vm_l^i} \neq \emptyset$. This can only be achieved if she requests launch of vm_l^i on a platform with profile SP_j. According to Proposition 2 and Proposition 3, such launch requests are rejected by DM_i; however, this does not prevent the $\mathcal{ADV}$ from attempting these options. The following two alternatives are available to the adversary:

a. The $\mathcal{ADV}$ launches $vm_l^i \mapsto CH_j$ on a platform under its own control (i.e. outside the provider domain).
b. The $\mathcal{ADV}$ launches $vm_l^i \mapsto CH_j$ on a valid platform in the provider network.

Option a: This option implies that the TL.AttestRequest step fails as shown in the proof of Proposition 1. In this case, the platform controlled by $\mathcal{ADV}$ does not get the symmetric key DK_i in return to the attestation request. Without access to DK_i, the only remaining option for the adversary is to attempt to break the final key request or the disc encryption scheme. Thus the following options are available:

- DBSP.DomKeyReq : The first option is to intercept a valid DBSP.DomKeyReq message for a storage domain $D_k^i \in \mathcal{D}^i_{vm_l^i}$ and replace the intercepted signature σ_{AIK} with her own own signature, σ'_{AIK} over the very same encrypted request (encrypted with a valid DK_i). However, similar to the earlier attempt to perform a TL.AttestRequest, this fails since the $\mathcal{ADV}$ does not have access to a valid attestation key. Any other attempt to send the adversary's own DBSP.DomKeyReq fails for the same reason.
- DBSP.DomKeyGen : The remaining option is to observe a valid DBSP.DomKeyGen for a domain $D_k^i \in \mathcal{D}^i_{vm_l^i}$ and attempt to access the encrypted storage keys. The latter fails due to the assumption of the TPM public key scheme soundness.
- *Attack Storage Encryption Scheme:* The remaining option for the $\mathcal{ADV}$ in this case is to directly break the disc encryption scheme. However, this is infeasible according to the disc encryption scheme soundness.

Option b: According to this option, the $\mathcal{ADV}$ tries launching vm_l^i using TL.Token on a platform with profile SP_j using its own credentials. The following impersonation alternatives are available:

- *Own token:* The adversary $\mathcal{ADV}$ sends a TL.Token message required by the protocol: $\mathsf{Enc}_{\mathsf{pk}_{\mathsf{TTP}}}\left(\tau \parallel H_1 \ \parallel H_2 \parallel SP_j \parallel idvm_l^i \parallel \mathcal{D}^i_{vm_l^i}\right), SP_j$, $\mathsf{pk}_{\mathcal{ADV}}, r, \sigma_{\mathcal{ADV}}$, where H_2 *either* is the hash of $\mathsf{pk}_{\mathsf{DM_i}}$ or the hash of $\mathsf{pk}_{\mathcal{ADV}}$. If the first option is used, the SC obtains in return to TL.AttestRequest, i.e. the TL.Attestation message, a sealed value with a hash $H_2' \neq H_2$ which causes the SC to abort the launch. If the second option is used, the complete launch procedure succeeds as expected. However, when the SC later requests the key for SR_i using the DBSP.DomKeyReq message, it includes the hash H_2 of the the $\mathcal{ADV}$ public key ($\mathsf{pk}_{\mathcal{ADV}}$) in the encrypted and signed request. $\mathcal{ADV}$ cannot change the hash value in this request unless she breaks the signature scheme of the request. Upon receiving the request, TTP identifies that $\mathcal{ADV}$ is *not* allowed to access $D_k^i \in \mathcal{D}^i_{vm_l^i}$ and does not return the storage keys in DBSP.DomKeyGen.
- *Legitimate token:* In this option, the $\mathcal{ADV}$ observes a valid c_1 in TL.Token for *another* vm with access rights to the intended domain and uses it to launch an own valid TL.Token message: $c_1, SP_j, \mathsf{pk}_{\mathcal{ADV}}, r, \sigma_{ADV}$. However, in this case the TL.AttestRequest fails as the profile in c_1 does not match the platform attested data. Furthermore, if the SC receives a reply to TL.AttestRequest, i.e. a TL.Attestation message, it would receive a sealed value with a hash $H_2' \neq H_2$, causing the SC to abort the launch. □

Proposition 5 (Domain Violation Attack). The DBSP protocol is sound against the domain violation attack.

Proof: Similar to the proof of Proposition 4, $\mathcal{ADV}$ has the following two options:

a. The $\mathcal{ADV}$ launches $vm_m^j \mapsto CH_j$ on a platform under its control (i.e. outside the provider domain).
b. The $\mathcal{ADV}$ launches $vm_m^j \mapsto CH_j$ on a valid platform in the provider network.

Option a: This option fails in analogy with the proof of Proposition 4, as $\mathcal{ADV}$ fails to successfully launch vm_m^j and her remaining options are to either attack the final key request or the disc encryption scheme, which both fail (see proof of Proposition 4).

Option b: In analogy with the proof of Proposition 4, $\mathcal{ADV}$ has only two options available: a full impersonation with an own chosen token of type $\mathsf{Enc}_{\mathsf{pk}_{\mathsf{TTP}}}\left(\tau \parallel H_1 \parallel H_2 \parallel SP_j \parallel idvm_m^j \parallel \mathcal{D}^j_{vm_m^j}\right)$, SP_j, $\mathsf{pk}_{\mathcal{ADV}}$, r, $\sigma_{\mathcal{ADV}}$, $\mathcal{D}^j_{vm_m^j} \subseteq \mathcal{D}_i$, or a partial impersonation reusing an observed c_1 of type $c_1, SP_j, \mathsf{pk}_{\mathcal{ADV}}, r, \sigma_{\mathcal{ADV}}$ for a subset of target storage domain. Both options fail in analogy with the arguments presented for the proof of Proposition 4. □

6 Implementation and Results

We next describe the implementation of the TL and DBSP protocols followed by experimental evaluation results.

6.1 Test bed Architecture

We describe the infrastructure of the prototype and the architecture of a distributed EHR system installed and configured over multiple VM instances running on the test bed.

6.1.1 Infrastructure Description

The test bed resides on four Dell PowerEdge R320 hosts connected on a Cisco Catalyst 2960 switch with 801.2q support. We used Linux CentOS, kernel version 2.6.32-358.123.2.openstack.el6.x86_64 and the OpenStack IaaS platform[5] (version Icehouse) using KVM virtualization support. The prototype IaaS includes one *"controller"* running essential platform services (scheduler, PKI components, SDN control plane, VM image storage, etc.) and three compute hosts running the VM guests. The topology of the prototype SDN

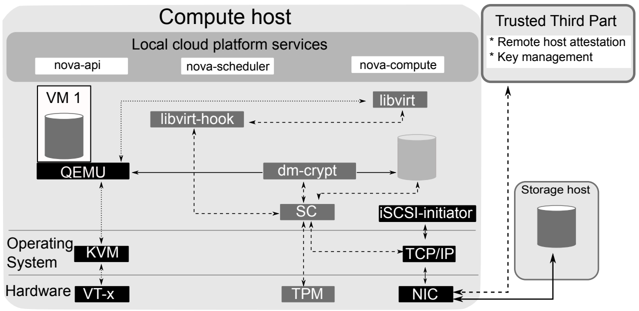


Fig. 4. Placement of the SC in the prototype implementation.

reflects three larger domains of the application-level deployment (front-end, back-end and database components) in three virtual LAN (VLAN) networks.

The compute hosts use libvirt[6] for virtualization functionality. We modified libvirt 1.0.2 and used the "libvirt-hooks" infrastructure to implement the SC for the TL and DBSP protocols. SC unlocks the volumes on compute hosts and interacts with the TPM and TTP (see Figure 4). It uses a generic server architecture where the SC daemon handles each request in a separate process. An inter process communication (IPC) protocol defines the types of messages processed by the SC. The IPC protocol uses sychronous calls with several types of requests for the respective SC operations; the response contains the exit code and response data. A detailed architecture of SC, including the main libraries that it relies on, is presented in Figure 5.

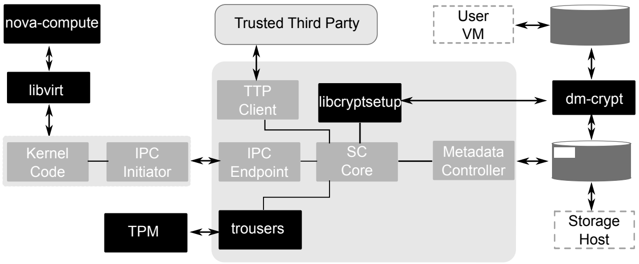


Fig. 5. Close-up view of the secure component implementation architecture, presented as a combination of components and existing libraries.

6.1.2 Application Description

The prototype also includes a distributed EHR system deployed over seven VM instances. This system contains one client VM, two front-end VMs, two back-end VMs, a database VM and an auxiliary external database VM. Six of the VM instances operate on Microsoft Windows Server 2012 R2, with one VM running the client application operates on Windows 7. The components of the EHR system communicate using statically defined IP addresses on the respective VLANS described in Section 6.1.1. Load balancing functionality provided by the underlying IaaS allots the load among front-end and back-end VM pairs. The hosts of the cluster are compatible with the TL protocol, which allows an infrastructure administrator to perform a trusted launch of VM instances on qualified hosts. Similarly, the infrastructure administrator can apply the DBSP protocol to protect sensitive information stored on the database servers.

TABLE 1
Overhead for unlocking a volume with DBSP (all times in ms)

Process	Event	Time
QEMU	Begin handle unlock request	0.083
SC	Requesting key from TTP	0.609
SC	Unseal key in TPM	2700.870
SC	Unlocking volume with cryptsetup	11.834
QEMU	End handle unlock request	26
	TOTAL	2714.004

6.2 Performance evaluation

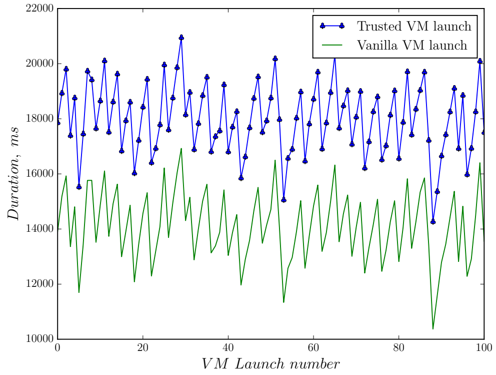


Fig. 6. Overhead induced by the TL protocol during VM instantiations.

Trusted launch: Figure 6 shows the duration of a VM launch over 100 successful instantiations: the TL protocol extends the duration of the VM instantiation (which does not include the OS boot time) on average by 28%. However, in our experiments we have used a minimalistic VM image (13.2 MB), based on CirrOS [7], while launching larger VM images takes significantly more time and proportionally reduces the overhead induced by TL.

DBSP Processing time: Table 1 shows a breakdown of the time required to process a storage unlock request, an average of 10 executions. Processing a volume unlock request on the prototype returns in ≈2.714 seconds; however, this operation is performed *only when attaching* the volume to a VM instance and does not affect the subsequent I/O operations on the volume. A closer view highlights the share of the contributing components in the overall overhead composition. Table 1 clearly shows that the TPM unseal operation lasts on average ≈2.7 seconds, or 99.516% of the execution time. According to Section 4.2, in this prototype we use TPMs v1.2, since a TPM v2.0 is not available on commodity platforms at the time of writing. Given that the vast majority of the execution time is spent in the TPM unseal operation, implementing the protocol with a TPM v2.0 may yield improved results.

DBSP Encryption Overhead: Next, we examine the processing overhead introduced by the DBSP protocol.

[5] OpenStack project website: https://www.openstack.org/
[6] Libvirt website: http://libvirt.org/
[7] CirrOS project website: https://launchpad.net/cirros

Figure 7 presents the results of a disk performance benchmark obtained using IOmeter[8]. To measure the effect of background disk encryption with DBSP, we attached two virtual disks to a deployed server VM described in 6.1.2. The storage volumes were physically located on a different host and communicating over iSCSI. We ran a benchmark with two parallel workers on the plaintext and DBSP-encrypted volumes over 12 hours. Next, we disabled in the host BIOS the AES-NI acceleration, created and attached a new volume to the VM and reran the benchmark. This has produced three performance data result sets: plaintext, DBSP encryption and DBSP encryption with AES-NI acceleration. Figure 7 summarises the *total IO*, *read IO* and *write IO* results. It is visible that the measurements *'4 KiB aligned (*DBSP*) with AES-NI'* and *'1 MiB (*DBSP*) with AES-NI'* are roughly on par with the plaintext baseline: *'4 KiB aligned'* and *'1 MiB'*. The performance overhead induced by background encryption is at 1.18% for read IO and 0.95% for write IO. We can expect that this performance penalty will be further reduced as the hardware support for encryption is improved. Disk encryption without hardware acceleration (*'4 KiB aligned (*DBSP*)'* and *'1 MiB (*DBSP*)'*) is significantly slower, as expected, with a performance penalty of respectively 49.22% and 42.19% (total IO). It is important to reemphasize that the runtime performance penalty is determined exclusively by the performance of the disk encryption subsystem. DBSP only affects the time required to unlock the volume when it is attached to the VM instance, as presented in Table 1.

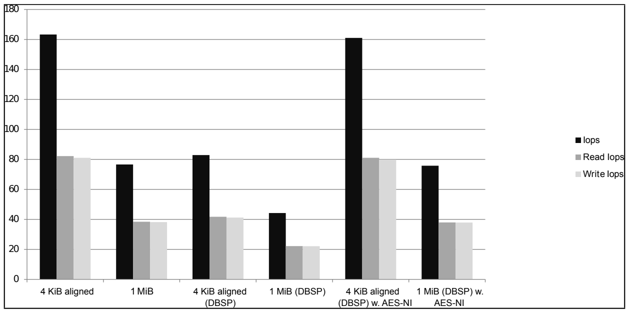


Fig. 7. Benchmarks results on identical drives: plaintext, with DBSP, with DBSP and AES-NI acceleration.

7 Application Domain

The presented results are based on work in collaboration with a regional public healthcare authority to address some of its concerns regarding IaaS security. We have deployed the prototype described in Section 6, further extended by integrating a medication database, and evaluated it through end-user validation and performance tests. Our results demonstrate that it is both *possible* and *practical* to provide strong platform software integrity guarantees to IaaS tenants and efficiently isolate their data using established cryptographic tools. Platform integrity guarantees allow tenants to take better decisions on both workload migration to the cloud and workload placement within IaaS. This contrasts with the current, "flat" trust model, where all IaaS hosts declare the same – but *unverifiable* for the tenant – trust level.

An essential conclusion of this practical exercise is that the additional cost of providing security guarantees can be *effectively offset* by composing cloud services from different competing providers, without having to delegate the trust among these providers. Thus, in our cloud model the tenant can purchase cheaper cloud disk storage without any additional risk for data confidentiality.

Another conclusion is that while organizations operating on sensitive data, e.g. public healthcare authorities, consider the risks of migrating data to IaaS clouds as unacceptable, the majority of available providers use commercial-off-the-shelf (COTS) cloud platforms with limited capabilities to enhance the security of their deployments, failing to meet the customer requirements. This demonstrates the need to incorporate integrity verification and data protection mechanisms into popular COTS cloud platforms by default. We hope that these important lessons will inspire new secure, usable and cost-effective solutions for cloud services.

On the practical side, specifically regarding the role of the TTP, we envision two scenarios. The TTP could either be managed by the tenant itself (for organizations with enough resources and expertise), or by an external organization (similar to a certificate authority). The first scenario allows the tenant to retain the benefits of cloud services along with additional security guarantees. Similarly, in the second scenario, smaller actors can obtain the same benefits without the need to invest into own attestation infrastructure. In both scenarios, in order to protect the cloud provider the TTP would only operate on a *physical slice* of the resources (i.e. a subset of compute hosts) that correspond to the respective tenant domains.

8 Conclusion

From a tenant point of view, the cloud security model does not yet hold against threat models developed for the traditional model where the hosts are operated and used by the same organization. However, there is a steady progress towards strengthening the IaaS security model. In this work we presented a framework for trusted infrastructure cloud deployment, with two focus points: VM deployment on trusted compute hosts and domain-based protection of stored data. We described in detail the design, implementation and security evaluation of protocols for trusted VM launch and domain-based storage protection. The solutions are based on requirements elicited by a public healthcare authority, have been implemented in a popular open-source IaaS platform and tested on a prototype deployment of a distributed EHR system. In the security analysis, we introduced a series of attacks and proved that the protocols hold in the specified threat model. To obtain further confidence in the semantic security properties of the protocols, we have modelled and verified them with ProVerif [33]. Finally, our performance tests have shown that the protocols introduce a insignificant performance overhead.

This work has covered only a fraction of the IaaS attack landscape. Important topics for future work are strengthening the trust model in cloud network communications, data geolocation [34], and applying searchable encryption schemes to create secure cloud storage mechanisms. Our results show that it is possible and practical to provide strong

[8] IOmeter project website: http://iometer.org

platform software integrity guarantees for tenants and efficiently isolate their data using established cryptographic tools. With reasonable engineering effort the framework can be integrated into production environments to strengthen their security properties.

References

[1] N. Santos, K. P. Gummadi, and R. Rodrigues, "Towards trusted cloud computing," in *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing*, HotCloud'09, (Berkeley, CA, USA), USENIX Association, 2009.

[2] J. Schiffman, T. Moyer, H. Vijayakumar, T. Jaeger, and P. McDaniel, "Seeding Clouds With Trust Anchors," in *Proceedings of the 2010 ACM Workshop on Cloud Computing Security*, CCSW '10, (New York, NY, USA), pp. 43–46, ACM, 2010.

[3] N. Paladi, A. Michalas, and C. Gehrmann, "Domain based storage protection with secure access control for the cloud," in *Proceedings of the 2014 International Workshop on Security in Cloud Computing*, ASIACCS '14, (New York, NY, USA), ACM, 2014.

[4] M. Jordon, "Cleaning up dirty disks in the cloud," *Network Security*, vol. 2012, no. 10, pp. 12–15, 2012.

[5] Cloud Security Alliance, "The notorious nine cloud computing top threats 2013," February 2013.

[6] A. Michalas, N. Paladi, and C. Gehrmann, "Security aspects of e-health systems migration to the cloud," in *the 16th International Conference on E-health Networking, Application & Services (Healthcom'14)*, pp. 228–232, IEEE, Oct 2014.

[7] B. Bertholon, S. Varrette, and P. Bouvry, "Certicloud: a novel tpm-based approach to ensure cloud IaaS security," in *Cloud Computing, 2011 IEEE International Conference on*, pp. 121–130, IEEE, 2011.

[8] M. Aslam, C. Gehrmann, L. Rasmusson, and M. Björkman, "Securely launching virtual machines on trustworthy platforms in a public cloud - an enterprise's perspective.," in *CLOSER*, pp. 511–521, SciTePress, 2012.

[9] A. Cooper and A. Martin, "Towards a secure, tamper-proof grid platform," in *Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on*, vol. 1, pp. 8–pp, IEEE, 2006.

[10] W. Wang, Z. Li, R. Owens, and B. Bhargava, "Secure and efficient access to outsourced data," in *Proceedings of the 2009 ACM workshop on Cloud computing security*, pp. 55–66, ACM, 2009.

[11] D. Song, E. Shi, I. Fischer, and U. Shankar, "Cloud data protection for the masses," *IEEE Computer*, vol. 45, no. 1, pp. 39–45, 2012.

[12] S. Graf, P. Lang, S. A. Hohenadel, and M. Waldvogel, "Versatile key management for secure cloud storage," in *Proceedings of the 2012 IEEE 31st Symposium on Reliable Distributed Systems*, pp. 469–474, IEEE Computer Society, 2012.

[13] N. Santos, R. Rodrigues, K. P. Gummadi, and S. Saroiu, "Policy-Sealed Data: A New Abstraction for Building Trusted Cloud Services," in *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, (Bellevue, WA), pp. 175–188, USENIX, 2012.

[14] A.-R. Sadeghi and C. Stúble, "Property-based attestation for computing platforms: Caring about properties, not mechanisms," in *Proceedings of the 2004 Workshop on New Security Paradigms*, NSPW '04, (New York, NY, USA), pp. 67–77, ACM, 2004.

[15] A. Sahai, "Ciphertext-policy attribute-based encryption," in *In Proceedings of the IEEE Symposium on Security and Privacy*, 2007.

[16] S. Kamara and K. Lauter, "Cryptographic cloud storage," in *Financial Cryptography and Data Security*, vol. 6054 of *Lecture Notes in Computer Science*, pp. 136–149, Springer Berlin Heidelberg, 2010.

[17] A. Sahai and B. Waters, "Fuzzy identity-based encryption," in *Advances in Cryptology–EUROCRYPT 2005*, Springer, 2005.

[18] S. Kamara and C. Papamanthou, "Parallel and dynamic searchable symmetric encryption," in *Financial Cryptography and Data Security*, pp. 258–274, Springer, 2013.

[19] N. Paladi, C. Gehrmann, M. Aslam, and F. Morenius, "Trusted Launch of Virtual Machine Instances in Public IaaS Environments," in *Information Security and Cryptology (ICISC'12)*, vol. 7839 of *Lecture Notes in Computer Science*, pp. 309–323, Springer, 2013.

[20] N. Paladi, C. Gehrmann, and F. Morenius, "Domain-Based Storage Protection (DBSP) in Public Infrastructure Clouds," in *Secure IT Systems*, pp. 279–296, Springer, 2013.

[21] P. Mell and T. Gance, "The NIST Definition of Cloud Computing," tech. rep., National Institute of Standards and Technology, 2011.

[22] C. Waldspurger and M. Rosenblum, "I/O virtualization," *Communications of the ACM*, vol. 55, no. 1, pp. 66–73, 2012.

[23] D. Dolev and A. C. Yao, "On the security of public key protocols," *Information Theory, IEEE Transactions on*, vol. 29, no. 2, 1983.

[24] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: A virtual machine-based platform for trusted computing," in *ACM SIGOPS Operating Systems Review*, vol. 37, ACM, 2003.

[25] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6, 2007.

[26] F. Zhang, J. Chen, H. Chen, and B. Zang, "Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pp. 203–216, ACM, 2011.

[27] G. Greenwald, "How the NSA tampers with US-made Internet routers," *The Guardian*, May 2014.

[28] S. Goldberg, "Why is it taking so long to secure internet routing?," *Communications of the ACM*, vol. 57, no. 10, pp. 56–63, 2014.

[29] A. Michalas, N. Komninos, N. Prasad, and V. Oleshchuk, "New client puzzle approach for dos resistance in ad hoc networks," in *Information Theory and Information Security (ICITIS), 2010 IEEE International Conference on*, pp. 568–573, Dec 2010.

[30] Trusted Computing Group, "TCG Specification, Architecture Overview, revision 1.4," tech. rep., 2007.

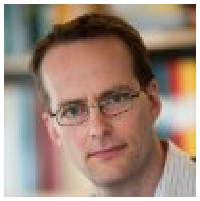
[31] B. Parno, J. M. McCune, and A. Perrig, *Bootstrapping Trust in Modern Computers*, vol. 10. Springer, 2011.

[32] P. Eronen and H. Tschofenig, "Pre-shared key ciphersuites for transport layer security (TLS)," 2005.

[33] B. Blanchet, "An efficient cryptographic protocol verifier based on prolog rules," in *Computer Security Foundations Workshop, IEEE*, pp. 0082–0082, IEEE Computer Society, 2001.

[34] N. Paladi and A. Michalas, ""One of our hosts in another country": Challenges of data geolocation in cloud storage," in *4th International Conference on Wireless Communications, Vehicular Technology, Information Theory and Aerospace Electronic Systems*, May 2014.

Nicolae Paladi is currently a PhD student at Lund University and researcher in the Security Lab at SICS. His research interests include distributed systems security with a special focus on cloud computing, infrastructure security, Internet security, virtualization and mobile platform security, trusted computing, as well as selected topics on privacy, anonymity and personal data protection.



Christian Gehramann is heading the Security Lab at SICS. The security lab has around 12 members and is performing applied research in security in virtualized systems, security for IoT, cryptography, authentication theory, security in cellular networks and on the Internet. Christian Gehrmann has conducted communication and computer security research for more then 20 years. He holds a PhD from Lund University and is also an associate professor at Lund University.

Antonis Michalas received his PhD in Network Security from Aalborg University, Denmark and later he worked as postdoctoral researcher at the Security Lab of SICS. Currently he is an assistant professor at the University of Westminster in London, UK. His research interests include private and secure e-voting systems, reputation systems, privacy in decentralized environments, cloud computing and privacy preserving protocols in participatory sensing applications.