Client-controlled Cryptography-as-a-Service in the Cloud

Sören Bleikertz[1], Sven Bugiel[2], Hugo Ideler[2],
Stefan Nürnberger[2], Ahmad-Reza Sadeghi[2]

[1] IBM Research - Zurich, Rüschlikon, Switzerland
sbl@zurich.ibm.com
[2] TU Darmstadt / CASED, Darmstadt, Germany
{sven.bugiel, hugo.ideler, stefan.nuernberger,
ahmad.sadeghi}@trust.cased.de

Abstract. Today, a serious concern about cloud computing is the protection of clients' data and computations against various attacks from outsiders as well as against the cloud provider. Moreover, cloud clients are rather limited in implementing, deploying and controlling their own security solutions in the cloud. The provider theoretically has access to stored keys in dormant images and deploying keys during run-time is infeasible because authenticating running VM instances is not possible.
In this paper, we present a security architecture that allows for establishing secure client-controlled Cryptography-as-a-Service (CaaS) in the cloud: Our *CaaS* enables clients to be in control of the provisioning and usage of their credentials and cryptographic primitives. They can securely provision keys or even implement their private virtual security module (e.g., vHSM or SmartCard). All clients' cryptographic operations run in a protected client-specific secure execution domain. This is achieved by modifying the Xen hypervisor and leveraging standard Trusted Computing technology. Moreover, our solution is legacy-compatible by installing a transparent cryptographic layer for the storage and network I/O of a VM. We reduced the privileged hypercalls necessary for administration by 79%. We evaluated the effectiveness and efficiency of our design which resulted in an acceptable performance overhead.

1 Introduction

Cloud computing offers IT resources, including storage, networking, and computing platforms, on an on-demand and pay-as-you-go basis. This promise of operational and monetary benefits has already encouraged various organizations to shift from a *"classical"* on-premise to a cloud-based service deployment of their workloads [12].

To secure those services, typically cryptographic security mechanisms are installed. Usually, these mechanisms require long-term secrets, e.g. SSL/TLS-secured web services need a secret key stored in the virtual machine (VM) for authentication purposes. Naturally, such long-term secrets are a valuable

target for attackers that compromise the client's service. In the classical on-premise datacenters, clients had the ability to incorporate security devices like Hardware Security Modules (HSMs) or SmartCards in order to protect their cryptographic credentials and operations. While this threat still holds in a cloud-based deployment [1,13,14], the difference is that the incorporation of security hardware is virtually impossible as cloud providers strictly prohibit physical customizations or access to their facilities. Additionally, outsourced data and computations are an easy prey for insider attackers at the provider's side since the client has willingly delegated control over his resources to the provider [25]. Controlling running instances of virtual machines, e.g. starting, stopping and maintaining them, is a necessity for every virtualization solution and is done by instructing the hypervisor from a privileged management domain which by default has ultimate access to all virtual machines. Insider attackers have access to this privileged domain and hence put clients' cryptographic credentials that are stored and processed in VMs at risk. This leads to trusting the cloud provider not to eavesdrop on the data. Consequently, it is desirable to build a cloud architecture that not only provides means to protect secrets even when the VM is exploited, but to also allow the client to deploy keys securely to the cloud without insiders being able to spy on it.

Cryptography-as-a-Service. In this paper, we present a security architecture that allows for provisioning *secret-less* client VMs in clouds and separating client's cryptographic primitives and credentials into a *client-controlled* and *protected* cryptographic domain (`DomC`). In contrast to other work that also advocates self-managed cloud services [9,39], we specifically built a solution that not only allows the establishment of a trust anchor and provisioning of user secret keys, but which also provides the protection of legacy VMs that were not tailored for our solution. We base our solution on the well-established concepts of a) segregating and encapsulating cryptographic operations and keys from the vulnerable client VM into a separate domain (`DomC`); and b) a trusted hypervisor that efficiently and effectively protects the separate `DomC` against a compromised or malicious management domain by subjecting it to the principle of least privilege. In contrast to related work, we overcome the aforementioned problem of actually deploying keys for use in the cloud. This requires novel security extensions to the VM life cycle management to protect the `DomC` during storage, transit, and instantiation, and to tightly couple it to the corresponding client's workload VM.

Contribution. We present the design and implementation of Cryptography-as-a-Service (*CaaS*), a solution to a practical security problem of clouds based on well-established and widely available technology. Our contributions are as follows:

- We present a dedicated, client-specific domain `DomC` for the client's cryptographic primitives and credentials that can be securely deployed with secrets by the client without the possibility for insiders or external attackers to gain access to them. Based on our security extensions to the hypervisor and well-established Trusted Computing technology, `DomC` can be protected from malicious insiders and outsiders in a reasonable adversary model. In

particular, we focus on integrating this protection in the entire VM life-cycle including deployment, instantiation, migration, and suspension.
– Clients can leverage their `DomC` in two different usage-modes: a) *Virtual Security Module* and b) *Secure Virtual Device*. Case a) emulates a virtual hardware security device, like an HSM/TPM, attached to the client VM while case b) interposes a transparent layer between the client VM and peripheral devices (disk or network) which encrypts all I/O data to/from those devices and hence protects *unaware* legacy OSes.
– We present the reference implementation of *CaaS* based on the Xen hypervisor and evaluate its performance for full disk encryption of attached storage and for a software-based HSM and its effectiveness with respect to different existing attack scenarios.
– Our modifications of the Xen hypervisor de-privilege the formerly privileged domain and separate former monolithic components into small, single-purpose and protected domains with a trusted computing base (TCB) that is orders of magnitudes smaller than the original version.

2 Model and Requirements

In the cloud service model hierarchy, we target the most general level *Infrastructure-as-a-Service* (IaaS) as depicted in Figure 1. In IaaS clouds, *Clients* rent virtual resources such as network and virtual machines from the provider and configure them according to their needs. Commonly, these VMs run public services such as web services offered to *End-Users* over the Internet.

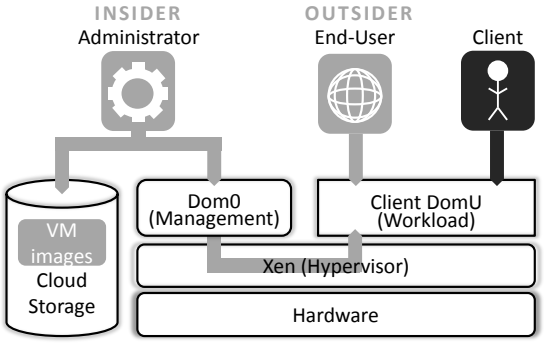


Fig. 1. Typical IaaS cloud model including our adversary and trust model.

We focus on the popular Xen hypervisor [3] and consequently use the Xen terminology. The clients' VM is denoted as `DomU`, meaning *unprivileged* domains that are guests on the hypervisor and have no direct hardware access. While there can be many `DomU` executing in parallel on the Xen hypervisor, there exists only one persistent *privileged* management domain, denoted `Dom0`. This

domain is usually not exposed to outsiders. Xen is a bare-metal hypervisor only concerned with the separation of virtual domains and their scheduling. It defers device emulation tasks to `Dom0`, that holds the necessary rights to access hardware resources. Thus, `Dom0` is naturally the place for the cloud infrastructure management software and their *Administrators* to operate in.

Besides computation, IaaS clouds normally also provide *Cloud Storage*. This storage is not just used for workload data but also to save the *VM images*, i.e., binary representations of VM states, from which `DomU`s are instantiated. In newer cloud usage models like cloud app stores [8], clients are also able to publicly provide their VM image and share it with other clients.

2.1 Trust Model and Assumptions

From a client's perspective, one of the most debated issues in cloud computing security is the trust placed in the cloud provider. In order to build a reasonable and practical trust model we do not assume a fully untrusted provider, but rather consider the involved actors and possible attacker types on the provider's side. We consider the following actors in our attacker model:

Compute Administrator. On a commodity hypervisor, `Dom0` and thus administrators, have read/write access to the memory of a running VM which is necessary for VM creation or, e.g., VM introspection. Hence, they are able to eavesdrop on data or even inject arbitrary code in the client's running VMs as shown by [25]. Thus, we do not trust the `Dom0`. We only consider attacks from administrators with *logical* access to the physical servers, e.g., by operating in the privileged management domain `Dom0`, and *not* attackers with physical access. This attacker model stems from practical scenarios, where datacenters are operated by a small team of trusted administrators with physical access and a large number of administrators with logical access, often outsourced and provided by third parties with limited trust.[3]

Storage Administrator. For administrators of storage resources, we consider an adversary that aims at learning cryptographic keys by inspecting or by modifying VM images, e.g., by injecting malicious code that will extract cryptographic keys at run-time. For storage administrators we allow physical access to hardware.

Network Administrator. We model the network administrators (omitted in Figure 1) according to the Dolev-Yao [16] attacker, i.e., the attacker has full control of the network and can eavesdrop and tamper with *all* network traffic.

Malicious Clients. It has been shown, that clients frequently store (and forget) security-critical information, such as cryptographic keys, in their public, shared VM images [8]. A malicious client can easily investigate those images and extract these information.

End-Users. Public (web-)services are a gateway for malicious intruders that compromise a VM, for instance, due to a vulnerability in the provided services.

[3] Note that purely cryptographic approaches [4,7,19] protect even against physical attacks. However, they are still impractical due to their enormous complexity overhead.

Hypervisor. We *exclude* run-time attacks on the hypervisor, as this is an open research problem and out of scope of this paper. Under this assumption, we consider a trustworthy hypervisor in the sense that the client can deploy mechanisms to verify the trustworthiness of the code a hypervisor is constituted of. This is accomplished using standardized trusted computing mechanisms such as *authenticated boot* and *remote attestation* [36] (cf. Section 3).

Denial-of-Service Attack. We *exclude* Denial-of-Service attacks from our model. This is motivated by the fact that the privileged domain `Dom0`, although not trusted, cannot be completely excluded from all operational and management tasks, and thus is always able to block correct operation.

2.2 Objectives and Requirements

Our main security objective is the protection of the client's cryptographic keys and operations in the cloud, similar to well-known SmartCards. We consider the following main security requirements to ensure the secure storage and usage of cryptographic credentials and operations in the client's VM:

1. Protection of long-term secrets of client VMs at runtime, i.e., an attacker who compromised the workload VM `DomU` or a malicious/compromised management domain `Dom0` cannot extract this information from the `DomU` VM.
2. The same must hold for the `DomU`'s integrity at rest, i.e., the client's dormant `DomU` VM image must be protected such that an attacker can neither extract credentials from it nor unnoticeably tamper with it.
3. Secure VM management operations, i.e., suspension and migration of the client `DomU` VM must preserve the integrity and confidentiality of `DomU`'s state on the source and target platform as well as during transit/storage.

3 Design and Implementation

In this section, we introduce the architecture and design decisions of Cryptography-as-a-Service (*CaaS*). The vital part of this paper is the deployment of secret keys to the secure environment `DomC`. In the first subsection 3.1, we explain the idea of our solution, followed by our security extensions to the hypervisor.

Prerequisites. We assume the availability of a hardware trust anchor on the cloud nodes in the form of a Trusted Platform Module (TPM). The TPM is used to securely attest the node's platform state [36]. For brevity, the following descriptions involve only one cloud client, however, we stress that the presented solutions can be easily applied to multiple client scenarios as well. Moreover, we apply the term *encryption* to abstractly describe a cryptographic mechanism for both confidentiality *and* integrity protection, i.e., authenticated encryption.

3.1 General Idea

Figure 2 illustrates the *CaaS* architecture using Xen. We achieve our goals by (1) severing the client's security sensitive operations and data in `DomU` into a *client-controlled* secure environment denoted `DomC`; (2) we degrade `Dom0` to an untrusted domain but retain it's purpose as administrative domain. This is achieved by extracting the domain management code (building, transferring, destroying VMs) and making this code run bare-metal in a new virtual machine. The resulting small trusted domain builder (`DomT`) then has exactly enough code and privileges to build new domains and makes the fully-blown management `Dom0` being a part of the TCB obsolete. Instead, `Dom0` now merely forwards commands to `DomT`. The necessary modifications in the Xen hypervisor are described in subsection 3.2.

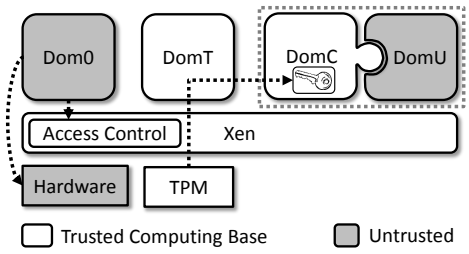


Fig. 2. Basic idea of *CaaS*: Establishment of a separate security-domain, denoted as `DomC`, for critical cryptographic operations.

To implement `DomC` and `DomT` as separate domains running on Xen without the need for a full-fledged operating system, we leveraged *Mini-OS* [34], which is a minimal stub domain directly interfacing with the Xen hypervisor. `DomC` exposes cryptographic library functions to the corresponding coupled workload VM (`DomU`) or automatically interposes external devices used by `DomU` to transparently encrypt/decrypt them. The privileged operations that traditionally would be done by `Dom0`, like domain building, domain migration etc. are segregated to a single-purpose stub domain `DomT`, the Trusted Domain Builder.

Usage Modes of DomC. Xen uses a *split driver* model for device drivers. It provides a front-end and a back-end module (cf. Figure 3). The latter controls the actual physical device while the former provides a virtualized representation of that device to VMs. In *CaaS*, we leverage this split-driver mechanism to connect `DomC` as a Xen virtual device to `DomU`. Figure 3 shows the two operation modes of `DomC` that we describe below: *Virtual Security Module* and *Secure Device Proxy*.

Virtual Security Module. In this mode of operation, `DomC` resembles a security module such as an HSM. In this mode, `DomU` has to be aware of the `DomC` so that it can use its interface for outsourcing traditional cryptographic operations like

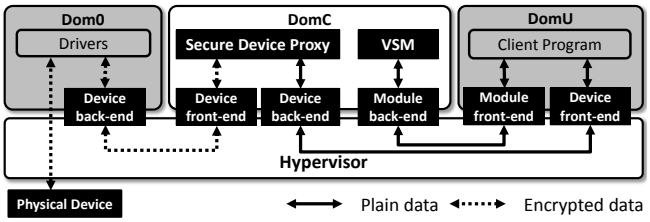


Fig. 3. Usage Modes: `DomU` can use `DomC` either as *Virtual Security Module* (VSM) or to secure its storage or network data with a transparent proxy.

an SSL/TLS wrapper for a web service running in the VM. In our prototypical implementation `DomC` emulates an HSM and provides a standardized PKCS#11-compliant interface for `DomU`.

Secure Device Proxy. In this mode, `DomC` acts as a transparent layer between `DomU` and external devices, such as attached storage medium or network card. We use this layer as a convenient building block for advanced applications such as booting fully encrypted VM images (cf. Section 3.3) or for legacy VMs that still want to profit from full-disk encryption. To achieve the pass-through, we chain two front-end-back-end communication channels. The first channel exists between `DomC` and `Dom0` where `DomC` connects to a device offered by `Dom0` (e.g., storage or network). The second channel exists between `DomC` and `DomU`, where `DomC` provides an identical device interface to `DomU`. `DomC` encrypts and decrypts on-the-fly all data in this stream. Although it is technically feasible that `DomC` writes directly to the physical device, routing encrypted I/O streams through `Dom0` avoids implementing (redundantly) device drivers in each `DomC`.

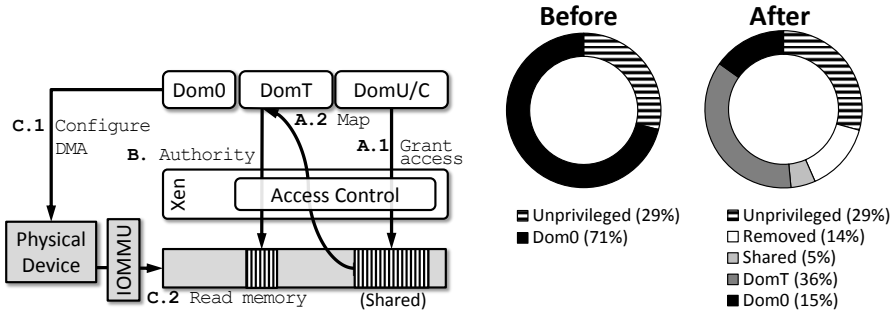
Both modes are not mutually exclusive. A transparent encryption layer can be used while `DomU` is yet aware of the `DomC` and additionally uses it for explicit cryptographic operations.

3.2 Security Extensions to the Xen Hypervisor

While the above mentioned modes seem not to require any changes to the Xen hypervisor, default Xen does not prevent `Dom0` from reading/writing another VM's memory. To prevent that, we added security extensions to the Xen hypervisor:

1. Additional Mandatory Access Control for low-level resources (e.g., memory) to isolate the client's `DomC` from any other domain including `Dom0` (Fig. 4(a)).
2. The binary privileged/unprivileged hypercall scheme was made more fine-grained to drastically de-privilege `Dom0` and to support certain hypercalls only for certain domains, namely `DomT` and `DomC` (Fig. 4(b)).

In default Xen, different mechanisms to access foreign memory of other domains exist (cf. Figure 4(a)):



(a) Additional access control. (b) Hypercall disaggregation.

Fig. 4. Access Control and disaggregation modifications of our Xen version.

Privileged Domains. In default Xen, `Dom0` is always able to map the memory pages of another domain since it needs to set up a new domain's memory before it is running. In order to remove this privilege, we separated the domain building functionality into `DomT`. To this extent we ported the Xen domain managment library `libxl` to Mini-OS to reside in `DomT`. Additionally, Xen's binary privileged/unprivileged hypercall scheme which allowed `Dom0` to map arbitrary foreign memory needed to be refined in order to support different domains with different privileges. This new access control is enforced in the logic of the Xen hypervisor for mapping foreign memory pages into a domain's memory range by extending the Xen Security Module (XSM) accordingly. The privilege of `Dom0` to access foreign memory is then disabled in the hypervisor while `Dom0` needs to forward domain management requests (building, migrating, destroying) to `DomT` which has now memory authority (step `B`). The concept of disaggregating code from `Dom0` was pioneered by Murray et al. [24] and enhanced in our design.

Grant tables. Grant tables are the default mechanism for establishing shared memory pages between different domains (e.g. for split drivers). The owning domain can discretely grant access to its memory pages to other domains (step `A.1`), which are then able to map these shared pages into their own memory space (step `A.2`). In *CaaS*, no additional access control on Grant Tables is required, as `DomU` and `DomC` are in control of their own pages and thus can by default deny any access from other domains.

IOMMU. A potential security risk are physical devices featuring Direct Memory Access (DMA), having access to the entire physical memory. DMA is configured by the domain that is in control of the physical hardware (by default `Dom0`; step `C.1`). We require hardware support in the form of an IOMMU (step `C.2`) to exclude the whole VM and Xen memory from the DMA range.

3.3 Detailed Image Setup Workflow

To ensure that the client can entrust her secrets and images to the cloud, we leverage standard Trusted Computing protocols for the Trusted Platform Module (TPM) [36]. This technology provides the means to establish a trusted end-to-end channel since the client can encrypt data such that only a platform in a certain trusted state S (i.e., running our modified version of Xen) is able to decrypt this data. Technically, this is realized using a TPM certified binding key (sk_{TPM}, pk_{TPM}) where the secret key sk_{TPM} is bound to the platform state S. The certificate *cert* proves that the key-pair was created by a genuine TPM and hence the binding property holds. To make the same key available on all cloud nodes, we use *migratable* keys, i.e., its usage is bound to one or more trustworthy platform states but not a particular platform. For brevity, we omit the setup of this TPM key from our protocol and refer to related work [10]. An *authenticated boot* [35] measures the platform state, during boot. Moreover, we make use of a TPM feature called *locality* to ensure that only the trusted hypervisor (i.e. not `Dom0`) is able to use the certified binding key sk_{TPM} and to further allow `Dom0` to still use the TPM, however, not at the locality reserved for the hypervisor. The pseudocode in algorithm 1 depicts the setup process of the client and trust establishment in detail.

Algorithm 1 Pseudocode for Setup Steps

1: get $(cert, pk_{TPM})$ from cloud node
2: **if** VALIDATE$(cert, pk_{TPM})$ **then**
3: $k \leftarrow$ GENERATESYMMETRICKEY()
4: $domCimage \leftarrow$ CREATECUSTOMDOMCIMAGE()
5: INJECTKEY$(domCimage, k)$
6: $enc_u \leftarrow$ ENCRYPT$(domUimage, k)$
7: $enc_c \leftarrow$ ENCRYPT$(domCimage, pk_{TPM})$
8: $ID \leftarrow$ UPLOADANDREGISTER(enc_c, enc_u)
9: **end if**

After the client verified pk_{TPM} using the certificate *cert* (line 2), she generates at least one new secret k (line 3) and securely injects that secret into her local plaintext image of `DomC` (line 5). `DomC` is able to act as transparent cryptographic protection (e.g., encryption) of an attached block storage (*Secure Device Proxy* mode) or as a SmartCard using key k. The `DomU` image is encrypted under k (step 6) and the configured `DomC` image is encrypted under pk_{TPM} (line 7) which constitutes the trusted channel explained earlier. Both encrypted images are then uploaded and registered in the cloud under a certain *ID* (line 8). Using *ID*, the client can manage her images, e.g., launch an instance from her `DomU` image.

3.4 Detailed Launching Workflow

The instantiation of the uploaded encrypted `DomU` image can be divided in two steps as shown in Figure 5: First, and only once after booting our modified Xen,

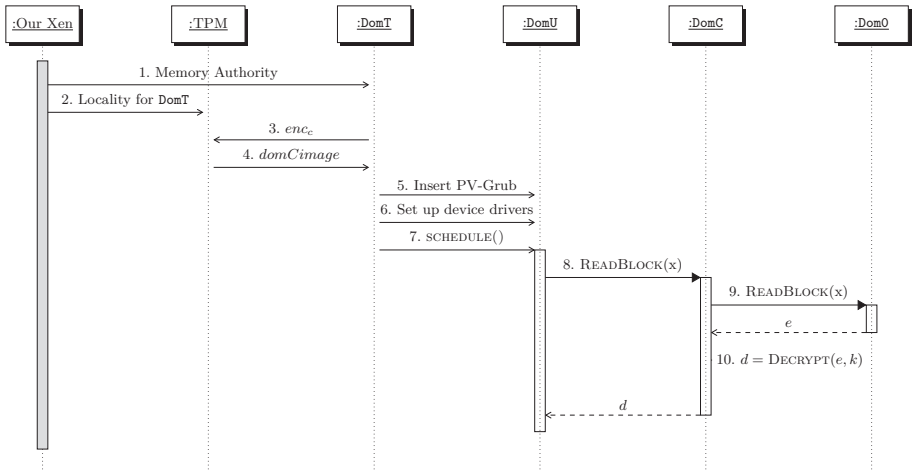


Fig. 5. Booting DomU and coupling with corresponding DomC

DomT is started with memory authority for the purpose of domain creation (step 1). Additionally, the locality of the TPM is set up in such a way that DomT is the only one allowed to use sk_{TPM} (step 2). DomT uses this sk_{TPM} to decrypt the DomC image *domCimage* with the aid of the TPM[4] (steps 3 & 4). DomT inserts the Xen bootloader PV-Grub[5] into the still pristine DomU image which is necessary for DomU to be able to boot from a device offered by DomC (step 5). Then, the front-end devices (cf. Figure 3) are set up to be available to PV-Grub to boot from (step 6). Once DomU is scheduled for the first time (step 7) and tries to read a block from the attached virtual disk (step 8), it gets transparently routed through DomC which reads the actual sectors from the traditional disk provided by Dom0 and decrypts them for DomU (steps 9 & 10).

Suspension and Live Migration. In order to support live migration, the standard Xen migration protocol needs to be wrapped but in essence works unaffectedly from the perspective of the client and DomU. Since we ported the Dom0 Xen interface (libxl) to DomT, the live migration request in Dom0 is simply forwarded to DomT which has access to any DomU's memory. DomT then migrates a running VM on-the-fly by first attesting the target host's integrity using its certificate *cert* and by piecemeal transmission of the memory content to the new trusted target host. Instead of migrating plaintext VM memory from one Node to another, the memory must be encrypted, since migration requires the involvement of Dom0 and a potentially untrusted network.

[4] The use of asymmetric cryptography in the TPM is an abstraction. Technically, the decryption using a TPM is more involved and requires wrapping a symmetric key with the pk_{TPM}/sk_{TPM} pair.

[5] http://wiki.xen.org/wiki/PvGrub

To restore the transferred state on the target platform, DomC has to be migrated as well in order to decrypt the migrated DomU state on the target platform. Restoring a VM state requires platform-dependent modifications to the state, such as rebuilding the memory page-tables. DomT's domain building code performs these modifications on DomU during DomU's resumption. Afterwards the new DomC is able to decrypt and resume the DomU state on the target platform and the old DomC on the source platform can be discarded. To achieve the protection of the transferred DomC state, this state is encrypted under the TPM key pk_{TPM}. Thus, only a target node running our trustworthy hypervisor is able to decrypt and resume the DomC state. We need to make sure that the version of our trusted Xen is not run outside of a trusted datacenter, e.g. our partly trusted cloud provider. For the sake of simplicity, in our proof-of-concept implementation we only allowed to migrate to other secure hosts that are within the same class-C-network. In case of suspension, the protocol works identical, except that the "target platform" is cloud storage to which the protected DomC and DomU states are saved by DomT.

4 Security

In this section we discuss how our architecture protects the client's cryptographic keys with regard to the requirements and adversary model defined in Section 2. We also discuss the corner cases that our architecture does not handle.

Compute Administrator. Our solution protects against a malicious *Compute Administrator*. This is guaranteed by the logical isolation of domains by the trusted hypervisor and the de-privileged management domain in which the administrators operate. Extracting the domain building process to DomT combined with the TPM based protocols (cf. Section 3.2) ensures that Dom0 cannot access DomT, DomC's or DomU's memory in plaintext. We empirically verified the mitigation of known attacks to extract confidential information from VMs [25].

Any modifications Dom0 does on the encrypted images during launch will lead to integrity verification failures and abortion of the launch, and hence form a denial-of-service. The same holds for the saved, encrypted state of DomU and DomC during migration and suspension. As mentioned in our adversary model, we exclude compute administrators with physical access, since it seems there exists no practical solution against these attacks yet.

Storage Administrator. Our solutions protects against a malicious *Storage Administrator* by storing images only in encrypted and integrity protected form. Thus, this attacker cannot extract any sensitive information from the images and any modification to the images before loading them into memory results in a denial-of-service attack. Solutions against replay attacks of outdated images, which we do not consider in this paper, can also be based on the TPM [29,37].

Network Administrator. Images and VM states are protected (encrypted and integrity checked) during provision to the cloud, transfer between cloud nodes and storage during migration and suspension, respectively. Thus, a malicious *Network Administrator* cannot extract the client's keys from intercepted network

data. However, dropping network traffic or tampering with it will lead to a denial-of-service attack. Freshness of network communications to protect against replay attacks or injection of non-authentic data is easily achieved by using message nonces or by establishing session keys.

End-Users. If an external attacker gains full (i.e. root) access to `DomU`, the attacker can misuse `DomC` as an oracle, e.g., to sign arbitrary messages in the client's name. This problem also applies to HSMs. A common countermeasure is an auditing mechanism within `DomC` that detects misuse based on heuristics (e.g., usage thresholds). The secrets however remain protected in `DomC`.

Malicious Clients. Since keys are neither stored nor processed within a customer VM, there is no risk of accidentally sharing them in public VM images. Thus, our solution protects against *Malicious Clients*, who inspect shared public VM images for credentials.

Adherence. Due to our isolation from the management domain, the cloud provider can no longer monitor the client's behaviour. This is a potential invitation to hide malicious/criminal activities such as providing illegal content. Other solutions [9] tackled this issue by installing a mutually trusted observer for the client VM's activities, which simultaneously preserves the client VM's privacy and checks the client's activity for conformance.

5 Performance Evaluation

We evaluated the performance overhead induced by offloading cryptographic operations to `DomC` for both the Secure Device Proxy and Virtual Security Module modes. Our test machine is a Dell Optiplex 980 with an Intel QuadCore i7 3.2GHz CPU, 8GB RAM, and a Western Digital WD5000AAKS - 75V0A0 hard-drive connected via SATA2.

Secure Device Proxy. This setup consists of the Xen v4.1.2 hypervisor with our extensions, an Arch Linux `Dom0` (kernel 3.2.13), a Debian `DomU` (kernel 3.2.0) and a Mini-OS based `DomC` and `DomT`. All domains and the hypervisor execute in 64-bit mode and each guest domain has been assigned one physical core. `DomU` and `DomT` have been assigned 256 MB of RAM while each `DomC` runs with 32 MB. All I/O data streams from `DomU` to the virtual block storage are passing through `DomC` and are transparently encrypted using AES-128 in CBC-ESSIV mode based on code ported to Mini-OS from the disk-encryption subsystem *dm-crypt* of the Linux kernel. We measure four scenarios:

Traditional. Standard Xen setup without an interposed `DomC` and no encryption.

dm-crypt in `DomU`. This extends the *Traditional* scenario with AES-128 CBC-ESSIV mode encryption of I/O data in `DomU` using dm-crypt.

`DomC` pass-through. This scenario interposes `DomC` between `DomU` and `Dom0` to merely pass-through I/O without encryption.

`DomC` (AES-128). This scenario extends the pass-through scenario with AES-128 CBC-ESSIV en-/decryption in `DomC`.

In a traditional Linux running as DomU, block device buffering is used for reads and writes, where writes occur asynchronously. In this setup, the performance overhead was negligible. To give a worst-case scenario, in this throughput benchmark we measure the induced performance overhead with all caching disabled and additionally only read/write random sectors to avoid hard disk buffer effects. The bandwidth measurements were taken using the fio tool[6] in the `DomU`. For each of the aforementioned four combinations measurements were taken with each read or write lasting exactly 10 minutes (see Figure 6). Performance measurements with asynchronous I/O, disk buffers left on (default Linux settings) and linear reads produced almost negligible overhead but had a high standard deviation.

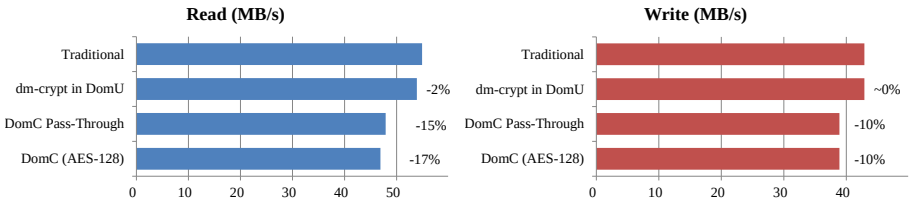


Fig. 6. Disk throughput performance

Virtual Security Module. Our setup consists of *SoftHSM*[7], a software-based implementation of a HSM that can be accessed via a PKCS#11 interface. We compare two scenarios: **a)** where *SoftHSM* is running in a Linux-based `DomC`, and **b)** when running inside a `DomU` and being accessed directly. In scenario **a**, the server resides within `DomC` and the client in `DomU`, and the communication is realized through our backend-frontend Virtual Security Module interface. In scenario **b**, both server and client reside in `DomU` and the network loopback device is used.

We measure the performance of RSA signing using an HSM. This is a typical scenario found in practice, e.g., CAs signing TLS certificates or signing of domain names within the DNSSEC system. In particular we are focusing on the latter scenario and leverage the benchmark software `ods-hsmspeed` from the OpenDNSSEC project[8]. As parameters for `ods-hsmspeed`, we selected 8 threads requesting signatures from the HSM, RSA1024 as the signing algorithm, and varying number of total signatures requested ranging from 1 to 10000.

Our results are illustrated in Figure 7. When requesting a low number of signatures, i.e., only 1 or 10, the costs for the connection and benchmark setup are more profound. However in practical scenarios, we expect a large number of signatures that are requested. Comparing the performance in terms of signatures per second between a *SoftHSM* residing in `DomU` vs. `DomC`, we notice a less than 3% overhead when offloading the cryptographic operations to `DomC`.

[6] FIO disk benchmark – `http://freecode.com/projects/fio`
[7] `http://www.opendnssec.org/softhsm/`
[8] `http://www.opendnssec.org/`

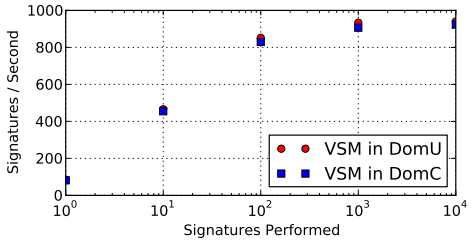


Fig. 7. Comparing the signing performance of a software-based HSM residing in `DomU` vs. `DomC`.

6 Related Work

The field of cloud security is very active and touches various research areas. In this section, we compare our *CaaS* solution to the closest related work.

Trusted Computing. In physical deployments, cryptographic services are typically provided by cryptographic tokens [2], hardware-security modules [17], generic PKCS#11-compliant modules, e.g. smart cards, and the Trusted Platform Module (TPM) [36]. In our approach, we study how such cryptographic services can also be securely provided in virtualized form in cloud deployments.

To provide TPM functionality to virtual machines, virtual TPMs have been proposed [5,28] and secure migration of VM-vTPM pairs by Danev et al. [15]. Our *CaaS* is conceptually a generalized form of such as a service, since `DomC` could also provide a vTPM daemon. However, in contrast to [5], our solution does not rely on a security service running within a potentially malicious `Dom0`.

Providing a cryptographic service over a network has been considered in large-scale networks, such as peer-to-peer or grid systems, by Xu and Sandhu [40]. Berson et al. propose a *Cryptography-as-a-Network-Service* [6] for performance benefits, by using a central service equipped with cryptographic hardware accelerators. Our *CaaS* targets specifically multi-tenant cloud environments and aims at tightly but securely coupling the client and her credentials to enable advanced applications such as transparent encryption of storage.

Different cloud architectures that rely on trusted computing have been proposed that ensure protected execution of virtual machines. The Trusted Cloud Computing Platform (TCCP) [31] by Santos et al. and the architecture proposed by Schiffman et al. [33] use TCG remote attestation to prove the trustworthiness of the cloud's compute nodes. Our approach also builds on Trusted Computing technology but with the goal to protect cryptographic operations and credentials from external and internal attackers. Santos et al. extended their TCCP architecture to address the problems of binary-based attestation [32] and data sealing with an approach very similar to property-based attestation [27].

Virtualization Security. Research that advocates the benefits of virtualization technology for security purposes has a long-standing history, even decades before the advent of cloud computing [20,21,26], and has introduced concepts that establish secure (virtual) execution environments [11,18]. They implement the concept of moving the security management to the virtualization layer by providing two different execution security contexts for VMs on top of a trusted VMM. Our architecture differs from those in that we provide client-controlled cryptographic primitives for multi-tenant virtualized environments (such as clouds) and thus have to tackle the challenges of how to securely provision and use those primitives in the presence of a malicious cloud management domain.

Other related works leverage nested virtualization to advocate similar goals as *CaaS*. Williams et al. introduced the *Xen-Blanket* [39], which adds an additional virtualization layer, empowering clients to avoid cloud provider lock-in. The *CloudVisor* [41] architecture by Zhang et al. adds a small hypervisor beneath the Xen hypervisor to protect client's `DomU` against an untrusted or compromised VMM or `Dom0` (including encrypted VM images). However, nested virtualization induces an unacceptable performance overhead and usually requires introspection. In *CaaS*, we *avoid* nested virtualization and instead apply Murray's concept of `Dom0` disaggregation [24] on top of the commodity Xen hypervisor, which is assumed trustworthy. We note, that hardening hypervisors against attacks is an active, orthogonal research area [38] from which our solution benefits.

The closest related work to ours, is the *Self-Service Cloud* (SSC) framework by Butt et al. [9], which was developed independently and in parallel to our work. In SSC, clients are able to securely spawn their own meta-domain, including their own user `Dom0`, in which they are in control of deployed (security) services, such as `DomU` introspection, storage intrusion detection, or storage encryption. This meta-domain is isolated from an untrusted `Dom0` using a mandatory access control framework in the Xen hypervisor (XSM [30]). In contrast to SSC, our *CaaS* takes care of client-controlled cryptographic operations and builds the basis for the actual key provisioning. We tackle the challenge of how to protect and securely use our `DomC`, running isolated but tightly coupled to its `DomU`. This requires modifications to the VM life cycle management, i.e., secure migration/suspension of `DomU` and instantiating fully encrypted `DomU` images.

Secure Execution Environment. Instead of relying on the trustworthiness of the virtualization layer, `DomC` would ideally run in a Secure Execution Environment (SEE) that is available as a hardware security extension on modern CPUs, e.g., *Flicker* by McCune et al. [23]. However, invocations of SEE suffer from the critical drawback that they incur a significant performance penalty. Consequently, this makes them unsuitable for streaming operations such as encryption of data of arbitrary length. McCune et al. address this issue with their *TrustVisor* [22] by leveraging hardware virtualization support of modern platforms, trusted computing technology, and a custom minimal hypervisor to establish a better performing SEE. Conceptually, TrustVisor is related to our *CaaS* from the perspective of isolating security sensitive code in an SEE. However, TrustVisor is

designed to protect this code from an untrusted legacy OS while *CaaS* targets the specific scenario of cloud environments and thus faces more complex challenges: First, *CaaS* has to address an additional virtualization layer to multiplex multiple clients' VMs. Second, our adversary model must consider a partially untrusted cloud provider and malicious co-located clients.

7 Conclusion and Future Work

In this paper we present the concept of secret-less virtual machines based on a client-controlled Cryptography-as-a-Service (*CaaS*) architecture for cloud infrastructures. Analogously to Hardware Security Modules in the physical world, our architecture segregates the management and storage of cloud clients' keys as well as all cryptographic operations into a secure crypto domain, denoted `DomC`, which is tightly coupled to the client's workloads VMs. Extensions of the trusted hypervisor enable clients to securely provision and use their keys and cryptographic primitives in the cloud. `DomC` can be used as virtual security module, e.g., vHSM, or as a transparent encryption layer between the client's VM and e.g. legacy storage. Furthermore, these extensions protect `DomC` in a reasonable adversary model from any unauthorized access that tries to extract cryptographic material from the VM – either from a privileged management domain or from outside the VM. The flexible nature of `DomC` allows for building more advanced architectures, such as Trusted Virtual Domains [10], on top of our CaaS. Evaluation of full disk encryption with our reference implementation showed that `DomC` imposes a minimal performance overhead. Future work aims at methods to mitigate run-time attacks against `DomU`, which enable an attacker to misuse the securely stored credentials. An avenue to mitigate this issue would be to install usage quotas heuristics in order to detect misuse. Further, secure logging in `DomC` would support post-misuse analysis.

Acknowledgments

This research has been supported by the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement n°257243 (TClouds project: http://www.tclouds-project.eu).

References

1. AlertLogic. An empirical analysis of real world threats: State of cloud security report. `http://www.alertlogic.com/resources/state-of-cloud-security-report/`, 2012.
2. R. Anderson, M. Bond, J. Clulow, and S. Skorobogatov. Cryptographic processors – a survey. *Proceedings of the IEEE*, 94(2):357–369, Feb. 2006.
3. P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *19th ACM symposium on Operating systems principles (SOSP'03)*. ACM, 2003.

4. A. Ben-David, N. Nisan, and B. Pinkas. FairplayMP: a system for secure multi-party computation. In *15th ACM conference on Computer and communications security (CCS'08)*. ACM, 2008.
5. S. Berger, R. Cáceres, K. A. Goldman, R. Perez, R. Sailer, and L. van Doorn. vtpm: virtualizing the trusted platform module. In *15th conference on USENIX Security Symposium*. USENIX, 2006.
6. T. Berson, D. Dean, M. Franklin, D. Smetters, and M. Spreitzer. Cryptography as a Network Service. In *Network and Distributed Systems Security Symposium (NDSS'01)*, 2001.
7. D. Bogdanov, S. Laur, and J. Willemson. Sharemind: A framework for fast privacy-preserving computations. In *13th European Symposium on Research in Computer Security: Computer Security (ESORICS'08)*. Springer, 2008.
8. S. Bugiel, S. Nürnberger, T. Pöppelmann, A.-R. Sadeghi, and T. Schneider. AmazonIA: When Elasticity Snaps Back. In *18th ACM Conference on Computer and Communications Security (CCS'11)*. ACM, Oct 2011.
9. S. Butt, H. A. Lagar-Cavilla, A. Srivastava, and V. Ganapathy. Self-service cloud computing. In *19th ACM Conference on Computer and Communications Security (CCS'12)*. ACM, October 2012.
10. L. Catuogno, A. Dmitrienko, K. Eriksson, D. Kuhlmann, G. Ramunno, A. Sadeghi, S. Schulz, M. Schunter, M. Winandy, and J. Zhan. Trusted Virtual Domains–Design, Implementation and Lessons Learned. *First International Conference on Trusted Systems (INTRUST)*, 2009.
11. X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. *ACM SIGOPS*, 2008.
12. Y. Chen and R. Sion. To cloud or not to cloud?: musings on costs and viability. In *2nd ACM Symposium on Cloud Computing (SOCC'11)*. ACM, 2011.
13. CVE-2007-4993. Bug in pygrub allows guests to execute commands in dom0.
14. CVE-2008-1943. Buffer overflow in xensource allows to execute arbitrary code.
15. B. Danev, R. J. Masti, G. O. Karame, and S. Capkun. Enabling secure VM-vTPM migration in private clouds. In *27th Annual Computer Security Applications Conference (ACSAC'11)*. ACM, 2011.
16. D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.
17. J. G. Dyer, M. Lindemann, R. Perez, R. Sailer, L. van Doorn, S. W. Smith, and S. Weingart. Building the IBM 4758 secure coprocessor. *IEEE Computer*, 2001.
18. T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: a virtual machine-based platform for trusted computing. In *19th ACM symposium on Operating systems principles (SOSP'03)*. ACM, 2003.
19. C. Gentry. Fully homomorphic encryption using ideal lattices. In *41st annual ACM symposium on Theory of Computing*. ACM, 2009.
20. N. Kelem and R. Feiertag. A separation model for virtual machine monitors. In *IEEE Computer Society Symposium on Research in Security and Privacy*, pages 78 –86, May 1991.
21. S. E. Madnick and J. J. Donovan. Application and analysis of the virtual machine approach to information system security and isolation. In *Workshop on virtual computer systems*. ACM, 1973.
22. J. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB reduction and attestation. In *IEEE Symposium on Security and Privacy (SP'10)*. IEEE, 2010.

23. J. McCune, B. Parno, A. Perrig, M. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *3rd European Conference on Computer Systems (EuroSys'08)*. ACM, 2008.
24. D. G. Murray, G. Milos, and S. Hand. Improving xen security through disaggregation. In *4th Int. conference on Virtual execution environments (VEE'08)*. ACM, 2008.
25. F. Rocha and M. Correia. Lucy in the sky without diamonds: Stealing confidential data in the cloud. In *41st International Conference on Dependable Systems and Networks Workshops (DSNW'11)*. IEEE, 2011.
26. J. M. Rushby. Proof of separability: A verification technique for a class of a security kernels. In *5th Colloquium on International Symposium on Programming*. Springer, 1982.
27. A.-R. Sadeghi and C. Stüble. Property-based attestation for computing platforms: caring about properties, not mechanisms. In *Workshop on New security paradigms (NSPW'04)*. ACM, 2004.
28. A.-R. Sadeghi, C. Stüble, and M. Winandy. Property-based TPM virtualization. In *11th International Conference on Information Security (ISC'08)*. Springer, 2008.
29. A.-R. Sadeghi, M. Wolf, C. Stüble, N. Asokan, and J.-E. Ekberg. Enabling fairer digital rights management with trusted computing. In *10th International Conference on Information Security (ISC'07)*. Springer, 2007.
30. R. Sailer, T. Jaeger, E. Valdez, R. Caceres, R. Perez, S. Berger, J. L. Griffin, and L. v. Doorn. Building a mac-based security architecture for the xen open-source hypervisor. In *21st Annual Computer Security Applications Conference (ACSAC'05)*. IEEE, 2005.
31. N. Santos, K. Gummadi, and R. Rodrigues. Towards trusted cloud computing. In *Hot topics in cloud computing (HotCloud'09)*. USENIX, 2009.
32. N. Santos, R. Rodrigues, K. P. Gummadi, and S. Saroiu. Policy-sealed data: A new abstraction for building trusted cloud services. In *21nd USENIX Security Symposium*. USENIX, 2012.
33. J. Schiffman, T. Moyer, H. Vijayakumar, T. Jaeger, and P. McDaniel. Seeding clouds with trust anchors. In *ACM workshop on Cloud computing security (CCSW'10)*. ACM, 2010.
34. S. Thibault. Stub domains: A step towards dom0 disaggregation. `http://www.xen.org/files/xensummitboston08/SamThibault_XenSummit.pdf`, 2010.
35. Trusted Computing Group (TCG). TCG specification architecture overview (revision 1.4), 2007.
36. Trusted Computing Group (TCG). Trusted platform module specifications, 2008.
37. M. van Dijk, J. Rhodes, L. F. G. Sarmenta, and S. Devadas. Offline untrusted storage with immediate detection of forking and replay attacks. In *2007 ACM workshop on Scalable trusted computing (STC'07)*. ACM, 2007.
38. Z. Wang and X. Jiang. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *2010 IEEE Symposium on Security and Privacy (SP'10)*. IEEE, 2010.
39. D. Williams, H. Jamjoom, and H. Weatherspoon. The xen-blanket: virtualize once, run everywhere. In *7th ACM european conference on Computer Systems (EuroSys'12)*. ACM, 2012.
40. S. Xu and R. Sandhu. A scalable and secure cryptographic service. In *Data and Applications Security XXI*, volume 4602 of *LNCS*. Springer, 2007.
41. F. Zhang, J. Chen, H. Chen, and B. Zang. Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *23rd ACM Symposium on Operating Systems Principles (SOSP'11)*. ACM, 2011.