Enforcing Access Controls for the Cryptographic Cloud Service Invocation Based on Virtual Machine Introspection

Fangjie Jiang[1,2,3], Quanwei Cai[1,2(✉)], Le Guan[4], and Jingqiang Lin[1,2,3]

[1] Data Assurance and Communication Security Research Center, CAS, Beijing 100093, China
{jiangfangjie,caiquanwei,linjingqiang}@iie.ac.cn
[2] State Key Laboratory of Information Security, Institute of Information Engineering, CAS, Beijing 100093, China
[3] School of Cyber Security, University of Chinese Academy of Sciences, Beijing 100049, China
[4] Pennsylvania State University, State College, USA
lug14@ist.psu.edu

Abstract. Most cloud providers afford their tenants with cryptographic services that greatly escalate the protection of users' private keys. Isolated from the guest operating systems (OSes), the keys are kept confidential even if the OS kernel is compromised. However, existing cryptographic services are ineffective in the access control of these critical services. In particular, they enforce controls for the key accesses mainly based on non-cryptographic authentication/authorization information (i.e., the identity and the password). Some platforms leverage other information such as the resource identification of the Virtual machine (VM) (e.g., IP address). Therefore, once the password is leaked, the attacker could invoke the cryptographic service in the victim VM. Moreover, sophisticated attackers can exploit vulnerabilities in the guest OS kernel and stealthily invoke cryptographic services. In this paper, we propose a new scheme named En-ACCI to improve the security of cryptographic service invocation in the cloud and achieve better access controls as well as auditing by leveraging the rich VM context provided by virtual machine introspection (VMI). To the best of our knowledge, we are the first in the literature to discuss these security issues involved in the invocation of cryptographic services in the cloud. We address the challenges by using an access control mechanism atop a set of optimization to VMI. We have implemented a prototype of En-ACCI, and our evaluation demonstrates that En-ACCI effectively addresses the authorization and audit issues in the cloud-based cryptographic service and the introduced performance overhead is modest.

Keywords: Cryptographic service · Cloud security
Invocation security · VMI · Access control
Virtualization-based security

L. Chen et al. (Eds.): ISC 2018, LNCS 11060, pp. 213–230, 2018.
https://doi.org/10.1007/978-3-319-99136-8_12

1 Introduction

Cloud computing is becoming more and more popular due to its agility, elasticity, reliability, and scalability. Using cloud computing, an enterprise will reduce the investment in IT infrastructures and focus on their core business. They also greatly benefit from add-on services offered by cloud providers. For example, virtual cryptographic computing and key management services bring enhanced protections for the tenants' cryptographic keys.

In these cryptographic computation services, tenants do not store the private keys. Instead, the cloud providers keep the keys securely and provide interfaces for tenants to invoke the computations of signing/encryption/decryption. The cloud providers, as the delegate of the tenants, protect the private keys by storing them in physically-separated devices (e.g., hardware security module) and never leak them into unsecure environments. They may further block unauthorized invocations to the cryptographic services with strict access control policies. For instance, only an invoker possessing the correct identity and the corresponding password is authorized to execute the cryptographic computation. Due to the increasing security requirements from tenants, such add-on services have become one of the major competitions in the cloud computing market. Typical cryptographic computing services include AWS CloudHSM [3] and Alibaba Aliyun encryption service [1]. The key management service (KMS) includes AWS KMS [4], Alibaba Aliyun KMS [2], Microsoft Azure Key Vault [8], Tencent Tencentyun KMS [10], etc.

Compared with other add-on services, the cryptographic service is much more important because it is used to ensure the security properties (e.g., data confidentiality, authentication, etc.) of other services. The cloud providers can improve the security of cryptographic services in two ways. First, they provide secure storage for the cryptographic keys. This can be achieved by existing techniques such as data encryption or dedicated devices. Second, the cloud enforces strict access control policies to the cryptographic service invocation and implements audit mechanisms of these invocations (e.g., AWS CloudTrail [5]). The latter records the invoker's identity, the source IP address, the time, the requested operation and the parameters of the operation in the specified storage for further compliance checks, security analysis, and troubleshooting.

However, existing approaches seem to be insufficient. For example, existing access control implementations of cryptographic services [2,4,8,10] are based on the identity of invokers and the corresponding passwords, which may be leaked due to dishonest developers or operators. The monitor tools only record the account information, identity and IP address of VM, the requested operation information, and do not log the inner context of the VM. An adversary can easily circumvent such monitoring. In particular, the adversary having the correct identity and the password may trigger a malicious process in the victim VM that invokes the cryptographic service directly, or even remotely exploit existing process (e.g., through network connections or file handlers) for cryptographic service invocation.

We observe that existing defense leverages coarse-grained information, which cannot provide rich information about the context of service invocation. We envision a new defense that incorporates diverse in-context information to log the invocation of critical services. Except for the basic information such as identity and password, the cloud may also check the VM context. Combining these rich information can greatly raise the bar for the attacker to circumvent the audit system. For example, the tenant may specify a sound access control policy, allowing the cryptographic computation to be invoked only by legitimate processes in the whitelist under authorized uid at a specified time frame. Request for cryptographic computation is fulfilled only if the invoking process is not compromised and the information of the invoker comply with the predefined access control policy. On receiving the request, the cloud provider checks the identity, the password, and the inner context of the VM, before performing the cryptographic computation. In addition, the cloud also records the inner context of VM during computation, e.g., the processes, the network connections, opened files' information, account, requested operation information, VM's identifier and IP address, for a better audit.

There are two approaches to collect context information of service invocation. One is that the VM reports such information when interacting with the hypervisor. However, a malicious process could potentially manipulate the results submitted to the hypervisor, or even hijack the system calls to return false results. The other approach is based on VMI [17], which actively analyzes the VM state using memory forensic techniques. Although this method may also get the incorrect context information of service invocation, it is transparent to the VM. Also, it reads the memory of VM directly, so it is not subject to the attack which tampers with the results. Adopting VMI technology to collect the context information is promising. However, it still remains challenging:

- **Introspection timing.** Ideally, when the cryptographic service is invoked, the VMI components should be triggered immediately to check the VM state. However, VMI components are not tightly coupled with the programs in the VM that invoke cryptographic service. In order to be informed about the cryptographic events in time, we have two options. First, the VMI component is triggered only when the cryptographic service is invoked. In our scheme, the VMM invokes the VMI component when the cryptographic service request is received. Second, the VMI components monitor the whole VM continuously. Obviously, this would introduce considerable overhead.
- **Authenticity of the context of cryptographic services.** The VM kernel may be infected by rootkits. The rootkits may tamper with the invoker's process context, misleading the VMI components.

In this paper, we propose a new scheme named En-ACCI to improve the security of cryptographic service invocation in the cloud and achieve better audit by leveraging rich VM context provided by the VMI. The cloud returns the results of a cryptographic request only after checking that the VM is in a trustworthy state. Moreover, the cloud records the context information during crypto computing. En-ACCI innovates by adapting VMI for cryptographic service with

improved performance. Many schemes provide cryptographic service, such as KMS [2,4,8,10], virtio-ct [20], vTPM [18] and etc. KMS provides cryptographic service via https networks connections. In virtualization platform, VMM emulates network card. That is to say, all the network I/O operations of the VM are trapped into the VM monitor (VMM). Similar to KMS, virtio-ct implemented a software HSM. Each cryptographic service request also causes the VM to be trapped into the VMM. To address the aforementioned challenges, we need to modify the I/O handle module in the VMM. Once the I/O module finds that the VM-Exit event is related to the cryptographic service, En-ACCI will be triggered. To get the correct context information of the invoker, we use the VMI tool to bridge the semantic gaps. Moreover, before collecting the context information of VM, En-ACCI scans rootkit in the VM. We have implemented an En-ACCI prototype based on QEMU/KVM [7,9], and integrated it with a cryptographic service named virtio-ct [20] which stores the cryptographic keys in a dedicated storage and completes the cryptographic computation in the trusted VMM. The main contributions of En-ACCI are as follows:

- To the best of our knowledge, we are the first in the literature to discuss the security issues involved in the invocation of cryptographic service in the cloud environment. We propose to address the associated challenges by using a new access control mechanism atop virtual machine introspection techniques.
- We have developed a set of optimization to existing VMI techniques to improve the performance of En-ACCI.
- We have implemented a prototype of En-ACCI, and our evaluation demonstrates that En-ACCI effectively addresses the authorization and audit issues in the cloud-based cryptographic service and the introduced performance overhead is modest.

The rest of the paper is organized as follows. We introduce the background in Sect. 2 and describe the design of En-ACCI in Sect. 3, followed by the implementation in Sect. 4. In Sect. 5, we analyze the performance and security of En-ACCI. Related work is introduced in Sect. 6. Finally, we draw the conclusion in Sect. 7.

2 Background

In this section, we firstly give a description of Kernel-based Virtual Machine (KVM), a popular virtualization solution based on QEMU. Then, we introduce essential knowledge about the virtual machine introspection (VMI) [17] and Executable and Linking Format [15] to better understand our solution.

2.1 Kernel-Based Virtual Machine

The KVM [23] is a popular virtualization solution based on QEMU. Taking advantage of hardware-assisted virtualization extensions such as Intel VT and AMD-V, it supports executing guest intrusions naively in the host system, thus improving performance significantly. At a lower level, it is implemented as a

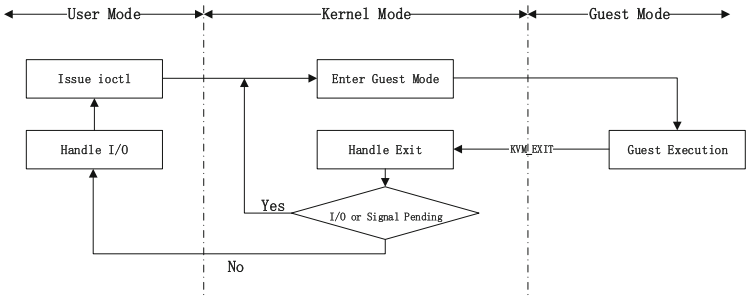

Fig. 1. Guest execution loop

loadable kernel module, which provides a set of `ioctl()` system calls to QEMU. QEMU, as a user-land process, is responsible for machine emulation, scheduling, resource allocation, and isolation etc. Therefore, QEMU can be viewed as a Virtual Machine Monitor (VMM).

In QEMU-KVM, except for the traditional kernel mode and user mode, another execution mode called guest mode is added, as shown in Fig. 1. The guest mode is essentially the mode in which the guest OS runs. When an exception or whatever critical system event configured by the KVM is caught, the VM exits, which is intercepted by the KVM module running in the kernel mode. Then KVM handles the exception either in the kernel mode directly and returns, or forwards it to the user mode. In the latter case, after handling the exception, QEMU calls another ioctl system call to resume the guest execution.

2.2 Virtual Machine Introspection

The Virtual Machine Introspection (VMI) is a technology in which the VMM dynamically inspects the execution context (internal state) of each VM. The purposes are mainly for security checking [17,19], software patching [13,14], and digital forensics [11]. To implement VMI functions, the VMM has to recover the semantic information of the inspected VM from the view of physical memory.

Depending on the VM OS, the VMI tools get the internal state of VM directly or reconstruct the high-level semantics. Installing a secret data collection module in the virtual machine [19,26] makes VMI tools effectively get the VM state. But there is a risk that the data collection module may be compromised. Utilizing the OS knowledge (e.g., system symbol map and kernel data struct) of the VM, VMI tools [11,17] reconstruct high-level semantics from the underlying binary data. And these VMI tools inspect the VM memory directly without installing an assistant module in the VM. VMM also obtains the state data of specific hardware devices of VM externally and then deduces high-level semantic information with the help of hardware architecture knowledge. Antfarm [22] proposed a hardware-based scheme to enable the VMM to track the processes and infer critical process events such as creation, context switch and exit.

2.3 Executable and Linking Format

This section describes an object file format called the Executable and Linking Format (ELF) [15], which is widely used by many OSes including Linux, Solaris, IRIX and OpenBSD. The proposed En-ACCI needs to access information stored in the EFL file to perform integrity check of the code segments. There are three types of ELF files: relocatable file, executable file and shared object file. All of them share a very similar format but are used for different purposes. As implied by the name, the executable file is directly loaded into the memory by OS for execution. The relocatable file contains code and data for individual program modules, which cannot be executed by its own, but needs to be statically linked with other object files to assemble an executable file. Similar to relocatable files, the shared object file is a module of the whole program. However, it is not statically linked during compilation. Rather, it is dynamically loaded into the memory by a dynamic loader implemented by relevant run-time.

Code and data in an object file are organized into sections, which are pointed to by a descriptive `section header table`. The sections are the basic modules which are linked together by the linker during linking process. In particular, sections with the same attributes are combined together to form a new section. Another important meta-data in an ELF file is the `program header table`, which provides program information for the loader. Therefore, it is mandatory for an executable file. A segment is essentially a chunk of aligned data/code with the same attributes. An entry in the `program header table` designates each segment's starting/ending addresses, the corresponding offset to the file image and attributes. The loader is responsible for loading the contents from file to the virtual memory.

3 System Design

In this section, we first present the threat model, followed by an overview of En-ACCI. Then, we detail our design from two aspects – access control policy and memory analysis methodology.

3.1 Threat Model

En-ACCI is designed to prevent unauthorized invocation to the cryptographic service. We consider an adversary model in which the VM OS is partially compromised by the attacker. Concretely, the attacker could invoke the cryptographic service by executing any user-space code in the VM. He could also conceal his unauthorized access to cryptographic service by leveraging rootkits that manipulate system log, process, network and file handle.

As with other VMI systems, we assume the integrity of the basic memory layer of an OS kernel. That is, the logical addresses of the kernel symbols and static kernel data structure are not modified [25].

In a cryptographic cloud service, the client invokes the cryptographic service through the provided program interface. We assume that the cryptographic key is

well protected by the provider, and the cryptographic computation is performed in a protected environment. The adversary cannot infer the cryptographic key from the provider through side channel attacks. Moreover, we assume the operators of the cryptographic service are honest and never leak the key, invoke the service nor modify the log information illegally. We assume that the design and implementation of cryptographic algorithms are secure. Finally, VMM is free of bugs. This is a very practical assumption considering the much smaller code base of VMM compared with full-fledged OSes.

3.2 Overview

A conceptual architecture of the proposed solution is illustrated in Fig. 2. As shown in the figure, En-ACCI is a software component in the VMM, which is non-bypassable when an app intends to invoke the cryptographic service. If the request is granted, En-ACCI forwards the request to the cryptographic service, which is securely implemented.

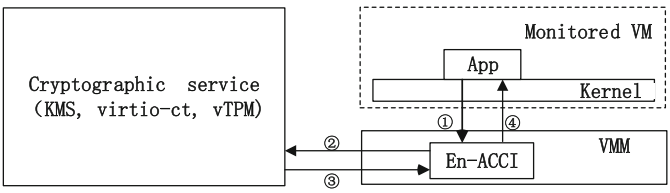

Fig. 2. En-ACCI architecture

To enforce security checking and auditing, we design our system to include the following processes.

- **Generating profile and hash meta-data.** For each version of OS, the operator needs to generate the corresponding profile, which includes the logic address of the essential kernel symbols and the offsets of elements in the kernel structure. This process is invoked only once for each version of the OS. The profile may be reused for different VMs with the same OS version and doesn't need to be regenerated when any kernel module is installed or removed. The tenant also needs to calculate the hashes of code segments in authorized binaries and relevant dynamic linking libraries. The hash meta-data include executable name, virtual addresses and length about the code segments. The hash meta-data are used to verify the integrity of the authorized processes.
- **Specifying access control policy.** The tenant may specify the access control policy for the service invocation on each cryptographic key. The access control policy defines the conditions that must be satisfied when the cryptographic service is invoked. The tenant may set the effective time for each policy and modify the policy when needed. In our implementation, the policy may specify the user ID, user group ID, name, start time, opened files

and established network connections of the process who may invoke the cryptographic service. Note that the list is flexible and can be extended in the future. The details of the access control policy are provided in Sect. 3.3.

- **Rootkit detection.** Once receiving the cryptographic service invocation, En-ACCI runs rootkit detector to check the state of the VM. Rootkit detector includes many VMI tools to check the critical kernel data and kernel text. If a rootkit is detected, the incident is reported.
- **Sampling and analyzing the VM memory.** Once receiving the cryptographic service request, VMM samples the necessary memory region of the VM and analyzes it to obtain the elements specified in the access control policy. We traverse the list of the processes in the VM and determine the process who invokes the cryptographic service according to the file handle or network connection corresponding to the cryptographic key.
- **Access control enforcement.** For each cryptographic service invocation, after sampling and analyzing the VM memory, we obtain the elements contained in the access control policy and compare the values of the elements obtained from the analysis with these defined in the policy. We fulfill the cryptographic service requests only when the policy is satisfied.

3.3 Access Control Policy

The tenant may define the access control policy based on different properties. These properties are extracted from the inner VM context. In the current version, En-ACCI supports the following access control properties.

- **PID, UID and GID**. After executing the application in the monitored VM, the tenant obtains the process identifier (PID), the identifier of the user that executes the application (UID), and the identifier of the corresponding user group (GID).
- **Process Name.** The name of the process is usually the name of the execution file, and is limited to 16 characters by default in Linux.
- **Location in the process tree.** The tenant may specify the blacklist and whitelist of the processes (according to PID, process name) that run concurrently in the VM or in the path from the init process to the application that invokes the service.
- **Process start time.** The tenant may obtain the accurate start time of the application externally, and specify it in the policy about the time frames during which cryptographic service can be invoked.
- **Opened file list.** The application may need to open a set of files in the different periods of normal execution. The cryptographic service is invoked in a critical region that should be well protected. Therefore, the tenant may specify the blacklist and whitelist of the files (through the file name and path) for this application during the cryptographic service invocation.
- **Allowed network connections.** Similar to the opened file list of the process, the tenant may specify the 5-tuple in the form of (src ip, src port, dst ip, dst port, protocol) of established connections during the service invocation.

The combination of the elements in the policy raises the bar for the adversary to invoke the cryptographic service without being noticed. Moreover, the tenant may find whether the VM contains suspicious processes or kernel modules that stealthily invoke the cryptographic service illegally, by comparing the results returned by the command with the ones logged in En-ACCI, and update the access control policy (e.g., forbidding the service invocation until the VM state is recovered) in time.

3.4 Memory Analysis

En-ACCI adopts memory analysis to sample the memory of the monitored VM, figure out the memory region related with the application that invokes the service, and obtain the information required by the access control policy.

Memory analysis is performed in VMM, which has access to the memory image of the monitored VM. According to the profile of the corresponding OS, we obtain the logical address of the first process (e.g., `init` in Linux). After transferring it to the address in the VMM address, we analyze it to get the semantic information, based on the offset (defined in the profile) of each element. With the semantic information of the first process, En-ACCI gets the addresses of all the processes in the VM, and obtains the their semantic information. En-ACCI figures out the memory region of the cryptographic service invoker by comparing the information of file list or networking connections with the ones corresponding to the cryptographic key handler. After figuring out the region corresponding to the service invoker, En-ACCI parses the figured memory region to obtain the semantic information, compares the parsed values with the ones specified in the policy, and finally returns the decision to the cryptographic service.

However, after obtaining the address of the process, the corresponding process may exit, which means that semantic information parse fails. In this case, we need to re-parse the previous process in the process chain to obtain the address of the cryptographic service invoker process again. The previous process may also exit before parsing. In the worst case, we need to trace back to the first process (i.e., `init` in Linux).

4 Implementation

We have implemented a prototype of En-ACCI based on QEMU-KVM v1.7.1. En-ACCI supports Linux distributions with kernel version 2.6 or above, and Windows OS as VMs. In the following, we describe our implementation on the CentOS v6.6 (Linux 3.13.7).

4.1 Framework Implementation

Figure 3 demonstrates the architecture of En-ACCI. We showcase how En-ACCI is used to protect virtio-ct [20], a cryptographic token implemented on KVM.

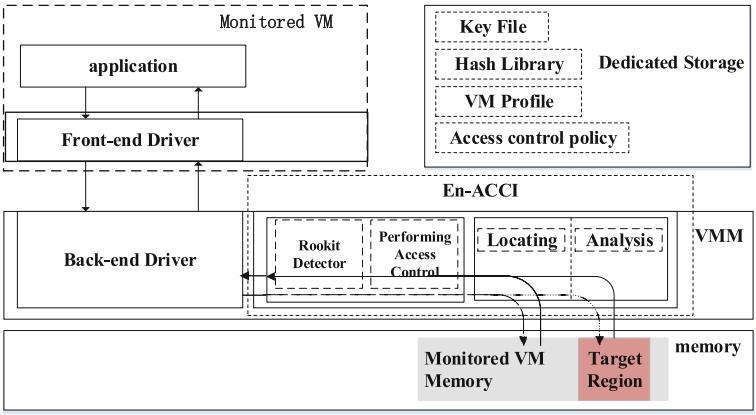

Fig. 3. The architecture of En-ACCI when integrating with virtio-ct [20].

In virtio-ct, the application in the monitored VM invokes the cryptographic service through a file handle, which is associated with the front-end driver. The parameters for the cryptographic computation include the identity of the cryptographic key, the corresponding PIN and the plaintext/ciphertext. The front-end driver routes these parameters to the virtio bus. The back-end driver in QEMU fetched the arguments on the bus and performs the corresponding cryptographic computation. The back-end driver invokes En-ACCI before performing the actual computation. En-ACCI conducts the following two steps. It first determines whether the VM is infected by rootkits, and then checks whether the inner context of the monitored VM satisfies the specified access control policy. More specifically:

- **Step 1:** Before performing the cryptographic computation, En-ACCI checks whether the request is invoked in a benign environment. This prevents the back-end driver from being abused by attackers, e.g., accessing keys without authorization or DoS attack by invoking intensive cartographic requests. Here, the rootkit detector sequentially invokes all the included tools to check (known) rootkits. To check whether the inner VM state satisfies the specified policy, En-ACCI first identifies the memory region of the process that invokes the cryptographic service. Regarding virtio-ct, we can locate the invoker based the opened files of the process, as the invoker needs to open the special file that represents the cryptographic key. Then, En-ACCI parses the identified memory region, and performs the access control according to the obtained semantic information. If any rootkit is detected or the access control policy is not satisfied, the back-end driver refuses to fulfill the cryptographic computation request.
- **Step 2:** Before returning the result to the front-end driver, En-ACCI performs checking to avoid the result to be obtained by the illegal user who comprises the VM or invoker process during the cryptographic computation.

In this step, En-ACCI reuses the result obtained in the previous step. For example, there is no need to identify the invoker's memory region. It directly invokes the rootkit detectors and performs access control by parsing the previously identified memory region. If the parse fails, or any rootkit exists, or the access control policy is not satisfied, the back-end driver refuses to return the computation results.

Except for virtio-ct [20], En-ACCI supports other types of the cryptographic cloud services. The major difference is the way to identify the corresponding cryptographic processes. In virtio-ct, we rely on the process's opened file information, which must include the corresponding the cryptographic key. For AWS CloudHSM and Alibaba Aliyun KMS, we identify a cryptographic process using the network connection information, because the process must maintain a network connection to the cryptographic service provider.

4.2 En-ACCI Implementation

Linux kernel adopts the data structure `task_struct` to store the metadata of a process. As shown in Fig. 4, the fields `comm`, `cred`, `pid` and `start_time` contain information about the process name, UID/GID, PID, process start time, while fields `files` and `fs` store information about open file information (including network connections) and filesystem information. Note that `tasks` is a list data structure used by Linux to keep all the processes in a linked list.

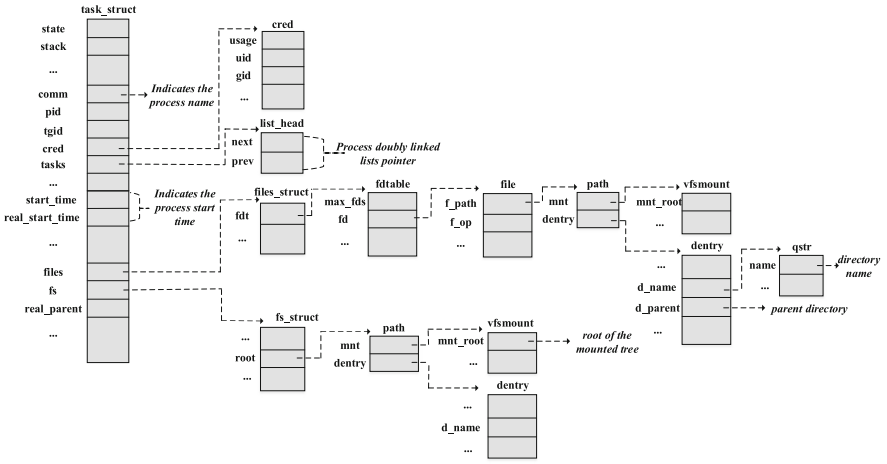

Fig. 4. The Linux process descriptor

We have generated an aforementioned profile For Linux 3.13.7. The profile records the logical address `init_task` and the offsets of the variables in `task_struct` shown in Fig. 4. En-ACCI adopts these information to parse the

semantic information of the `init` process and then follows the linked list indicated by the `tasks` field (offset is 1640) to obtain the logical addresses of the other processes. For each process, the PID, UID, GID, process name, and start time are obtained directly from the `task_struct`, while, En-ACCI needs to parse the fields `files` (of type `files_struct`) and `fs` (of type `fs_struct`) to infer the information of associated files. The variable `fs` provides the root dentry through the variable `root` (of type `path`). The variable `fdt` (of type `fdtable`) in the structure `files_struct` contains the information of all files opened by this process. That is, the member `fd` of structure `fdtable` lists the dentries for each opened file. The network connection is represented by a file in Linux. We distinguish it from normal files through the variable `f_op` in the structure `file`. The `f_op` points to the operation that can be invoked on this file, but points to the function `socket_file_ops` when the file is a socket.

The profile only provides the guest logical address of each symbol, while En-ACCI, being a component of VMM, can access the contents through guest physical address in the underlying host OS. Therefore, we need to perform the address translation. En-ACCI invokes `cpu_physical_memory_rw` provided by QEMU-Monitor to transfer the guest logical address to the guest physical address, and then access it for further parse. En-ACCI also records the address returned by `cpu_physical_memory_rw`, and uses it with the offset to access the memory in the host directly when no new guest logical address needs to be processed. We implemented the aforementioned procedure in a lightweight way. The whole implementation comprises about 700 lines of code.

The hash library provides the authorized process names, virtual addresses of the code segments when the authorized processes is loaded into memory and the digest of each code page. We analyze the authorized binary files offline in advance. We calculate the digest of each page in the code segments from start. For pages contributing to both code segments and data segments, we replace the data regions with all zero. Then we calculate the digest of each page and store them in the hash library. En-ACCI determines if the virtual page has been loaded into physical memory by `bit0` in the `page table entry` and verifies the integrity of the virtual page.

In our prototype, En-ACCI is integrated with virtio-ct [20], in which the application invokes the cryptographic service through a virtual device (i.e., a file). En-ACCI uses the information of this file to figure the memory regions corresponding to the application. The semantic analysis of this memory region and the access control process are performed twice, one before performing the cryptographic computation and the other before returning the result to the application. Enforcing the access control is implemented in less than 100 lines of code.

En-ACCI also logs the information for each cryptographic service invocation. In addition to the user identity, IP address and the geographical region of VM, the parameters and invocation time of the cryptographic service, it also includes the PID, UID, GID, process name, process start time, the parent process, opened files, established network connections, and the result of integrity checking.

For rootkit detector, we have implemented three VMI tools to detect a set of kernel rootkits that alter the control flow by modifying the critical kernel data

(e.g., IDT and system call table) or directly manipulate the kernel text. The three VMI tools check the integrity of IDT, system call table and kernel text respectively. Since our implementation is based on KVM, other popular VMI tools can be easily integrated in our framework.

5 Analysis

In this section, we evaluate the performance overhead introduced by En-ACCI and analyze the security improvement brought to cryptographic service invocations. Finally, we discuss the limitations of En-ACCI in our current prototype.

5.1 Performance Evaluation

The prototype of En-ACCI is implemented as a module for QEMU/KVM v1.7.1 using the C programming language. The profile of the target operating system is generated using memory forensics tool volatility [11] and dwarf-tools. We set up our evaluation environment with a Dell Optiplex 9020 powered an Intel i7-4770 CPU (3.4 GHz) and 16 GB RAM. We assigned 4 vCPUs and 4 GB RAM for the VMs. The host operating system and the VM run CentOS v7.0 and CentOS v6.6 respectively.

We compared the results obtained from En-ACCI with that obtained from the original virtio-ct and LibVMI-virtio-ct, which performs access control for virtio-ct based on LibVMI. LibVMI-virtio-ct is implemented as follows: on receiving the service invocation, it suspends the VM, invoking the LibVMI to analyze the memory for the entries specified in the access control policy, and returns the computation results to the resumed VM.

The performance was evaluated at different concurrency levels. In the experiment, we used an application with at most 8 threads to invoke 2048-bit RSA decryptions. The throughput is demonstrated in Fig. 5. Compared with the original virtio-ct, the throughput of En-ACCI is reduced by 17.77% (the number

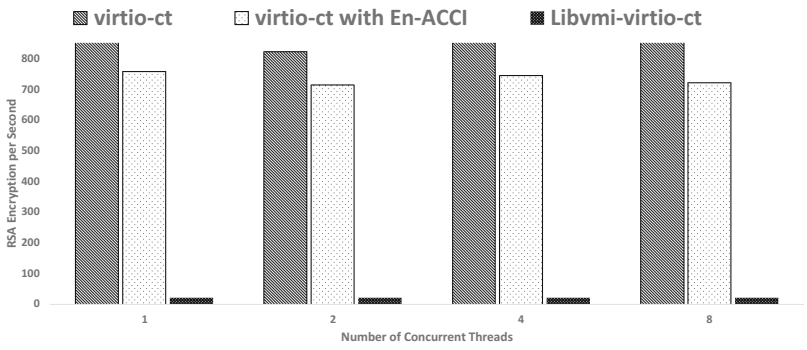

Fig. 5. Throughput of cryptographic service invocation.

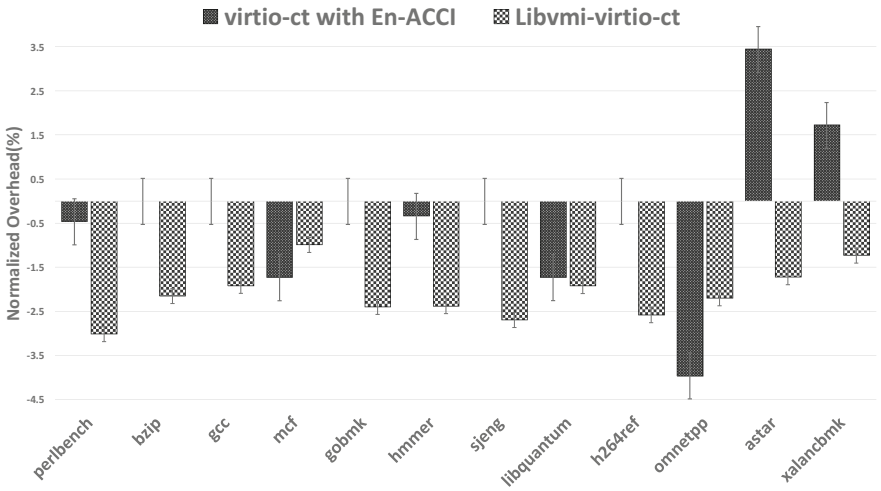

Fig. 6. SPEC INT 2006 perf. overhead.

of threads is 8), which is modest compared to 96.71% in LibVMI-virtio-ct (the thread number is 4).

We also ran SPECINT 2006 to evaluate the impact of En-ACCI to the other parallel processes. We set the performance of native virtio-ct [20] as the baseline, and compared En-ACCI with Libvmi-virtio-ct by running SPECINT 2006 when invoking the cryptographic service every 5 s with En-ACCI and Libvmi-virtio-ct integrated respectively. As shown in Fig. 6, the impact of En-ACCI to the other parallel processes is modest (less than 4.5%).

5.2 Security Analysis

En-ACCI raises the bar for the adversary to invoke the cryptographic cloud service from three aspects: 1. En-ACCI doesn't increase the risk of cryptographic key leakage itself, as it doesn't need the key for access control or audit. 2. En-ACCI enforces the access control policy specified by the tenant for each service invocation, which increases the difficulties of illegal invocation. 3. En-ACCI records the detailed information for each cryptographic cloud service invocation. This information is essential for causality analysis when the system is compromised.

For access control, in addition to identity and password used in KMS [2,4,10], En-ACCI leverages the approach described in Sect. 3.3 to extract inner VM state. En-ACCI checks user ID, user group ID and start time of the process to ensure the invoker process is created by the authorized user at the specified time. The attacker would fail to invoke the cryptographic service if the invoker process is started by a different account or is not started during the allowed time frames. En-ACCI compares the name of the process and digests of the process's code segment with that specified in the profile and hash library, to ensure only the

authorized executable program without integrity compromise can invoke the cryptographic service. The specified list of opened files and established network connections prevents the remote attacker who hijacks the victim process through file handles or network connections from invoking the cryptographic service.

In order to prevent kernel rootkit from manipulating the target memory region to fool En-ACCI, En-ACCI integrates existing rootkit detector. Once a rootkit is found, the cryptographic service is interrupted. Moreover, En-ACCI performs two steps for each service invocation; the first one prevents the attacker from abusing the cryptographic computation, while the second one ensures that the computation result is only returned to the legitimate invoker.

En-ACCI also has access to the detailed information about the inner VM state during the service invocation. Therefore, even if the access control mechanism is somehow bypassed (e.g., when the policy is incorrect), the cloud manager can identify the illegal invocation in time, analyze the malicious invocation thoroughly and modify the access control policy in time to avoid further malicious invocations.

The adversary who controls VM cannot compromise En-ACCI due to the isolation mechanism provided by the virtualization. Moreover, the source code of En-ACCI is only about 700 lines, which makes the formal analysis feasible.

5.3 Limitations

We discuss the limitations of the current En-ACCI prototype. En-ACCI relies on logical addresses of the kernel symbols and the kernel data structure to perform semantic analysis. However, as described in [12], the obtained semantic information may be incorrect when the kernel data structure is manipulated. In the current version, we rely on the followings assumptions to ensure the correctness of the obtained semantic information. (1) The guest OS is patched in time for known kernel-level vulnerabilities, which allow the attacker to hijack the control flow of the kernel. We admit that the attacker may still able to hijack the control flow using the zero-day vulnerabilities. (2) The integrity of the kernel is checked using existing rootkit detection tools. Therefore, our protection relies on the effectiveness of existing tools.

6 Related Works

In this section, we summarize existing work on secure service invocation and cryptographic key protection.

6.1 Cloud-Based Cryptographic Services

Virtio-ct emulates an HSM in VMM. Utilizing the isolation mechanism of the hypervisor, any code, including ring-0 malicious code in the guest OS cannot access cryptographic keys. To prevent the adversary from stealthily signing data, each time the key is accessed, virtio-ct drives the `pc-speaker` to make a sound

to notify the user. However, virtio-ct is designed for single PC scenarios instead of cloud.

For KMS [2,4,8,10], cloud providers provide access control strategy. KMS can be invoked in three ways in the current commercial design: (1) the web-based console (2) the command line interface and (3) the cloud service API. Although a variety of user authentication and access control are provided, the security of KMS ultimately depends on the identity management and password-based authentication mechanism. Audit is also provided in several providers. For example, AWSCloudTrail [5] can record very basic information of the cryptographic invocations.

6.2 Cryptographic Keys Protection

Various schemes have been proposed to protect the confidentiality of the cryptographic key against memory disclosure attacks. New features [6,16] provided by CPU manufacturer were adopted to protect the cryptographic keys. For example, Mimosa [21] only keeps plaintext of sensitive data in the transaction memory, which rolls back to the ciphertext once the memory is accessed by others. Intel Software Guard Extension (SGX) [6] allows user-level code to allocate private regions of memory, called enclave, which is isolated from the rest of the system, including OS and BIOS. With it, cryptographic services can be securely implemented.

As more and more services are being migrated to the cloud, the associated security problems emerge. To mitigate attacks from the inner VM, Virtio-ct [20] provides virtual cryptographic service while the corresponding key files are stored in the dedicated storage and the cryptographic program is executed in VMM. To prevent the attacks to VMM, TrustVisor [24] introduces a small hypervisor as TCB to enforce the data secrecy and program integrity, even if the OS is compromised.

En-ACCI adopts virtio-ct to prevent the attacks from the VM. It may also integrate the other works [6,21] to avoid the memory disclosure attacks on the physical machine when VMM is deployed, and adopt the schemes proposed in [24] to reduce the size of TCB where the cryptographic computation is performed.

7 Conclusion

We propose En-ACCI, a VMI-based mechanism to add another line of defense for cryptographic cloud services. En-ACCI enforces the access control for the cryptographic cloud service based on the rich VM context, and provides better audit by recording the detailed information of the VM and invoker process. To achieve better performance, instead of adopting existing VMI tools (e.g., libvmi) directly, En-ACCI analyses the VM's memory based on the logical addresses and the kernel data structure of the guest OSes, and further identifies and parses the memory regions of the invoker process in VMM. To ensure the correctness of the semantical information, En-ACCI integrates existing rootkit detection

tools and checks the integrity of the invoker's code segments. The performance evaluation demonstrates that the performance overhead caused by En-ACCI is modest (about 17.77%).

Acknowledgments. This work was partially supported by National Natural Science Foundation of China (No. 61772518).

References

1. Aliyun cryption service. https://www.aliyun.com/product/hsm
2. Aliyun Key Management Service. https://www.aliyun.com/product/kms
3. Amazon aws cloudhsm. https://amazonaws-china.com/cloudhsm/
4. Amazon aws key management service kms. https://amazonaws-china.com/kms/
5. Aws cloudtrail. https://amazonaws-china.com/cn/cloudtrail
6. Intel corporation, intel software guard extensions. https://software.intel.com/en-us/sgx
7. Kernel based virtual machine. http://www.linux-kvm.org/page/Main_Page
8. Microsoft key vault. https://www.azure.cn/home/features/key-vault/
9. Qemu open source processor emulator. http://wiki.qemu.org/Main_Page
10. Tencentyun key management service kms. https://cloud.tencent.com/product/kms
11. The volatility framework. https://code.google.com/archive/p/volatility/
12. Bahram, S., Jiang, X., Wang, Z., Grace, M., Li, J., Srinivasan, D., Rhee, J., Xu, D.: DKSM: subverting virtual machine introspection for fun and profit. In: 2010 IEEE Symposium on Reliable Distributed Systems, pp. 82–91 (2010)
13. Biedermann, S., Katzenbeisser, S., Szefer, J.: Leveraging virtual machine introspection for hot-hardening of arbitrary cloud-user applications. In: 6th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2014, Philadelphia, PA, USA, 17–18 June 2014 (2014)
14. Chen, P., Xu, D., Mao, B.: Clouder: a framework for automatic software vulnerability location and patching in the cloud. In: 7th ACM Symposium on Information, Compuer and Communications Security, ASIACCS 2012, Seoul, Korea, 2–4 May 2012, p. 50 (2012)
15. TIS Committee et al.: Tool interface standard (tls) executable and linking format (elf) specification version 1.2. TIS Committee (1995)
16. Intel Corporation: Chapter 8: Intel transactional memory synchronization extensions. In: Intel Architecture Instruction Set Extensions Programming Reference (2013)
17. Garfinkel, T., Rosenblum, M.: A virtual machine introspection based architecture for intrusion detection. In: Proceedings of the Network and Distributed System Security Symposium, NDSS 2003, San Diego, California, USA (2003)
18. Goyette, R.: A review of vTPM: virtualizing the trusted platform module. In: Proceedings of Network Security and Cryptography (2007)
19. Gu, Z., Deng, Z., Xu, D., Jiang, X.: Process implanting: a new active introspection framework for virtualization. In: 30th IEEE Symposium on Reliable Distributed Systems (SRDS 2011), Madrid, Spain, 4–7 October 2011, pp. 147–156 (2011)
20. Guan, L., Li, F., Jing, J., Wang, J., Ma, Z.: virtio-ct: a secure cryptographic token service in hypervisors. In: Tian, J., Jing, J., Srivatsa, M. (eds.) SecureComm 2014. LNICST, vol. 153, pp. 285–300. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23802-9_22

21. Guan, L., Lin, J., Luo, B., Jing, J., Wang, J.: Protecting private keys against memory disclosure attacks using hardware transactional memory. In: 2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, 17–21 May 2015, pp. 3–19 (2015)
22. Jones, S.T., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: Antfarm: tracking processes in a virtual machine environment. In: Proceedings of the 2006 USENIX Annual Technical Conference, Boston, MA, USA, 30 May–3 June 2006, pp. 1–14 (2006)
23. Kivity, A.: kvm: the linux virtual machine monitor. In: Linux Symposium, Ottawa, Ontario (2007)
24. McCune, J.M., Li, Y., Qu, N., Zhou, Z., Datta, A., Gligor, V.D., Perrig, A.: Trustvisor: efficient TCB reduction and attestation. In: 31st IEEE Symposium on Security and Privacy, S&P 2010, 16–19 May 2010, Berleley, Oakland, California, USA, pp. 143–158 (2010)
25. Seshadri, A., Luk, M., Qu, N., Perrig, A.: Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In: Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, 14–17 October 2007, pp. 335–350 (2007)
26. Vogl, S., Kilic, F., Schneider, C., Eckert, C.: X-TIER: kernel module injection. In: Lopez, J., Huang, X., Sandhu, R. (eds.) NSS 2013. LNCS, vol. 7873, pp. 192–205. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38631-2_15