virtio-ct: A Secure Cryptographic Token Service in Hypervisors

Le Guan[1,2,3(✉)], Fengjun Li[4], Jiwu Jing[1,2], Jing Wang[1,2], and Ziqiang Ma[1,2]

[1] State Key Laboratory of Information Security,
Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
{lguan,jing,jwang,zqma13}@is.ac.cn
[2] Data Assurance and Communication Security Research Center,
Chinese Academy of Sciences, Beijing, China
[3] University of Chinese Academy of Sciences, Beijing, China
[4] Department of Electrical Engineering and Computer Science,
The University of Kansas, Lawrence, USA
fli@ku.edu

Abstract. Software based cryptographic services are subject to various memory attacks that expose sensitive keys. This poses serious threats to data confidentiality of the stakeholder. Recent research has made progress in safekeeping these keys by employing isolation at all levels. However, all of them depend on the security of the operating system (OS), which is extremely hard to guarantee in practice. To solve this problem, this work designs a virtual hardware cryptographic token with the help of virtualization technology. By pushing cryptographic primitives to ring -1, sensitive key materials are never exposed to the guest OS, thus confidentiality is retained even if the entire guest OS is compromised. The prototype implements the RSA algorithm on KVM and we have developed the corresponding driver for the Linux OS. Experimental results validate that our implementation leaks no copy of any sensitive material in the "guest-physical" address space of the guest OS. Meanwhile, nearly 1,000 2048-bit RSA private requests can be served per second.

Keywords: Virtual cryptographic token · KVM · Virtio

1 Introduction

In computer and communications systems, data confidentiality is often provided by encryption. The strength of encryption mechanisms depend on both the security of the encryption algorithm and the secrecy of cryptographic keys. Over the years, there has been several encryption algorithms that withstand continuous cryptographic analyses. However, in a practical deployment, various reasons may cause the exposure of keys. For instance, key manager's dereliction of duty, improper software implementations, permeate attacks, etc. all put the keys at risk.

J. Tian et al. (Eds.): SecureComm 2014, Part II, LNICST 153, pp. 285–300, 2015.
DOI: 10.1007/978-3-319-23802-9_22

In particular, software vulnerabilities that leak memory regions to unauthorized subjects are main threats against the confidentiality of the keys. They extend to every level of the software stack and are often difficult to detect, due to the complication of computer systems. The sensitive data spread across the whole memory space (both kernel and user spaces) for longer than traditionally thought, even if the used memory is freed [1–3]. Applications designed with security in mind that zeros memory space containing the keys may also be fooled by the compiler, as such "superfluous" operations may be removed as a result of optimization [1]. Mobile-devices hoard a mass of sensitive information in plain-text in both RAM and stable storage. These data are very likely to be leaked once the device is lost [4]. The recently discovered OpenSSL vulnerability, namely Heartbleed [5], allows remote attackers without any privileged credentials to steal private keys. The bug is attributed to the loose inspection on the request packet that triggers a buffer over-read. Moreover, there are numerous subversions at the OS level that leak arbitrary kernel space memory to user space [6–8].

To migrate these attacks, threshold cryptosystems [9] and intrusion-resilient cryptosystems [10] are designed to withstand disclosure of some portion of the keys. In [11], keys are scrambled and dispersed in memory, but re-assembled in x86 SSE XMM registers when cryptographic computation is needed. Nikos et al. proposed implementing a cryptographic framework inside Linux kernel and providing cryptographic service to user space through a proven secure interface [12]. Unfortunately, all the above solutions only mitigate the problem to some extent. If the entire memory image is obtained by the attacker somehow, all "hidden" information is disclosed.

One of the most efficient and effective ways to safekeep keys is to employ hardware security modules (HSM). For example, Luna G5 is an usb-attached device that uses a dedicated chip to store user keys and perform cryptographic computation [13]. The keys are well protected both logically (keys cannot be legally accessed through the software interface) and physically (illegal physical invasion cannot obtain the keys). Optionally, whenever a cryptographic key is used, a LED flickers to alert the user that the key is being accessed. Since such approaches are secure and efficient, especially in insecure environments, they are often adopted in the industry, e.g., for authentication in online banking. However, hardware-based solutions all require additional costs, and are vendor dependent.

Motivation. We observe that hardware virtualization technology (VT-x for Intel and SVM for AMD) has been widely supported in commodity computers. By running a virtual machine, another layer of software isolation is added. The compromise of the guest OS does not harm the security the virtual machine monitor (VMM) or hypervisor. Based on this property, it is feasible to emulate a "HSM" in VMM for the guest OS such that arbitrary malicious code running in guest OS (including ring 0 malware) does not harm the key security.

In this paper, we present `virtio-ct`, a software virtual cryptographic token that aims to inherit some key advantages of a HSM. In particular, on top of virtualization technology, a `virtio-ct` virtual device:

- provides guest OS with RSA cryptographic service using emulated PCI interface.
- decouples cryptographic keys from the guest OS.
- is OS agnostic.
- audits all the accesses to the keys and emits physical signals in a mandatory way.

Like a real HSM, by designing an interface only for requests and responses, the cryptographic keys are never exposed to the guest OS. So even kernel compromise cannot retrieve the keys. At the same time, as `virtio-ct` is a software solution, users benefit from its flexibility, low costs and easy use.

The prototype is implemented on top of Kernel-based Virtual Machine (KVM) [14] and uses QEMU [15] as its user space device emulator. The communication channel is based on `virtio` [16], the de-facto standard for para-virtualization I/O. Experimental results show that `virtio-ct` achieves nearly 1,000 2048-bit RSA private key encryptions per second on an Intel core i7-4770S CPU.

Limitations. `virtio-ct` resembles a real HSM in many aspects, except that it is not resilient to physical attacks. A HSM defeats invasive physical attack by enclosing the key information in a tamper-sensing device. However, when a machine running `virtio-ct` services falls into an attacker's hand, this attacker could launch physical attacks to the RAM chip, for example, cold-boot attack [17].

`virtio-ct` only provides cryptographic key storage and computation services. A full Trusted Platform Module (TPM) also offers many other capacities like remote attestation [18]. Virtual TPM (vTPM) is a superset of `virtio-ct`. However, our work makes sense because of the following reasons: (1) vTPM needs a physical TPM to establish trust while `virtio-ct` does not rely on it. (2) A full vTPM is relatively error prone and may encounter many challenges because of its complexity. (3) System programmer needs expertise to work with TPM. In contrast, the application programming interface (API) of `virtio-ct` is much simpler: we provide services through the OpenSSL API by encapsulating several `ioctl` system calls.

Outline. The remainder of our paper is structured as follows: First, Sect. 2 introduces related works in this field. Then we describe the system model and applications of `virtio-ct` and clarify the attack model of it in Sect. 3. Next we give background information about VMM and `virtio` in Sect. 4. We introduce the design and implementation of `virtio-ct` in Sect. 5. Evaluations in terms of performance and security are presented in Sect. 6. Finally, Sect. 7 draws the conclusion.

2 Related Work

Protecting cryptographic keys from unauthorized access is a prerequisite for the security of the entire cryptographic system. Toward this, there are two generic ways. The first one keeps keys in the same memory space of the OS, while depends on software mechanisms to ensure strong isolation. Chow et al. proposed a secure deallocation mechanism to minimize the exposure of the keys [1]. Nikos et al. proposed a cryptographic framework inside Linux kernel to provide cryptographic service to user space through a proven secure interface [12]. Intel Software Guard Extension (SGX) solves the problem in hardware [19]. Specifically, it supplies new instructions to seal legitimate software inside an enclave and protected environment, irrespective of the privilege level of the malware.

`virtio-ct` falls into the second category, which escrows the key to a trusted third party. The keys are accessed either through network, PCI-E or usb interface in HSM solution. CleanOS [4] on the other hand, evicts the key that was used to encrypt sensitive data locally to the cloud when the data is not in active use. vTMP [18] emulates a TMP device in the cloud environment. It extended the current TPM V1.2 command set with virtual TPM management commands that allow users to create and delete instances of TPMs. Each created instance of a TPM holds an association with a VM throughout its lifetime on the platform. Compared with vTMP, `virtio-ct` provides a small yet important subset in the perspective of cryptographic computing. `virtio-ct` is more flexible (no dependance on real TPM) and efficient.

3 System Model and Threat Model

System Model. `virtio-ct` is designed to isolate cryptographic computation from the OS in commodity platform. For a secure conscious user, he or she would expect to run untrusted applications in an isolated environment to avoid possible infection to the host, while access sensitive data through a secure interface. To this end, the user may create a VM to execute that application while employ `virtio-ct` to request cryptographic service. Whenever the guest OS accesses the cryptographic keys, `virtio-ct` emits physical signals to notify the user, just like a real HSM.

Threat Model. We consider an attacker who can execute arbitrary code in the legacy OS – both in user space and kernel space. The attacker can achieve this by injecting customized code via buffer overflow attacks or implanting system level rootkits and Trojans. This implies that the attacker is able to manipulate page tables to access all desired memory regions.

We assume the attacker has no physical access to the computer. Otherwise, hardware-based attacks such as cold-boot attacks [17] or even bus-probing could be used to harm the security of the RAM chips.

We also assume that the underlying VMM is mostly safe. The compromise of the guest OS cannot escape from the VMM. That is to say, the adversary cannot

interfere with Virtual Machine Control Structures (VMCSs), Extended Page Tables (EPTs) and other sensitive structures that require higher privilege. Note that a trend in designing VMMs is that the code size is reducing. For example, the Xen hypervisor [20] has approximately 100 kilo lines of code (KLOC) while BitVisor [21] has only 20 KLOC. The result is that it is easier to verify the security of VMM itself.

4 Background

This section first gives an overview of several popular virtualization solutions, with detailed description of Kernel-based Virtual Machine (KVM). Then virtio, a de-facto standard for para-virtualization I/O devices is explained. `virtio-ct` is implemented on top of both solutions.

4.1 VMMs and KVM

VMM is a software layer that abstracts isolated virtual hardware platforms on which several independent guest OSes run in parallel. These virtual machines (VMs) share the same set of physical resources with VMM acting as resource manager and device emulator. VMM guarantees that VMs cannot access each other's resources, including memory regions.

With the prevalence of hardware virtualization extensions, most instructions of the guest OS can be executed directly in the CPU. Unlike those in the host OS, they are executed in a separated non-root mode. In this mode, certain predefined events that are considered risky can be intercepted by the VMM, for example, privileged instructions, interrupts and I/O instructions. In this way, VMM runs in a more privileged level because it can decide whether or not these guest-issued sensitive operations can be executed. In both modes, the executable can run in either of the four privilege levels defined in the x86 platform. So traditional OSes that employ separated privilege levels do not have to modify its code to accommodate the virtualized environment.

There are two types of VMM architectures. Type I VMMs run natively on the bare-metal hardware and implement all the VMM functions itself. This include Xen, VMware ESX/ESXi, Hyper-V, etc. On the contrary, Type II VMMs run in the context of the host OS. This simplifies the design of the VMM because many of the host OS's functionalities can be used readily. Most notable implantations among these are VMware Workstation, Oracle VirtualBox and KVM.

KVM. KVM is implemented as a loadable kernel module originally for Linux on the x86 platform, but later ported to PowerPC, System z (i.e., S/390) and ARM platforms. The KVM module is the core of the KVM solution. It initializes the CPU hardware and provides a serial of VMM management interfaces through the `ioctl` system call, for instance, creating a VM, mapping the physical address of the VM, and assigning virtual CPUs (vCPUs). A dedicated user space program, namely QEMU, provides for PC platform emulation and calls the KVM interface to execute guest OS code.

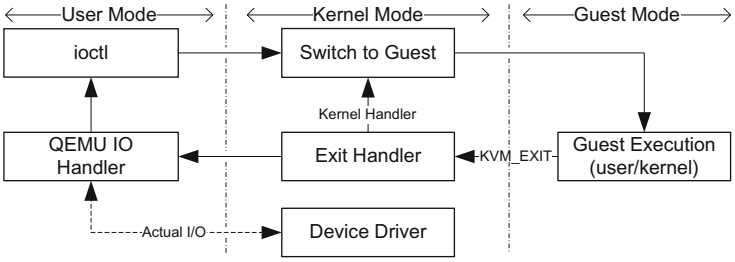

Fig. 1. Three vCPU modes in which the guest OS runs

In KVM, a guest OS is presented as a normal QEMU process and each vCPU is a thread. A QEMU process has three execution modes, namely, guest mode, kernel mode and user mode, as shown in Fig. 1. In guest mode, guest OS executes most of its instructions, either in user space or kernel space. Certain event causes a `VM-EXIT` and is intercepted by the KVM module. Then the QEMU process is in kernel mode. Based on the event that causes the trap, the exit handler deals with it inside the kernel or transfers it to user space. The former is called a lightweight exit while the latter is called a heavyweight exit, because the transfer leads to inter-ring switches. In user mode, QEMU accomplishes the exit handler and then calls the ioctl system call to resume the guest.

4.2 Virtio

In a full virtualization environment, the guest OS is unaware of being virtualized and requires no change of code. However, when encountering I/O operations, emulating hardware at the lowest level is inefficient. Conversely, in a paravirtualization environment, the guest and the VMM can work cooperatively to boost the I/O performance. Correspondingly, device drivers of guest OS should be modified (Fig. 2).

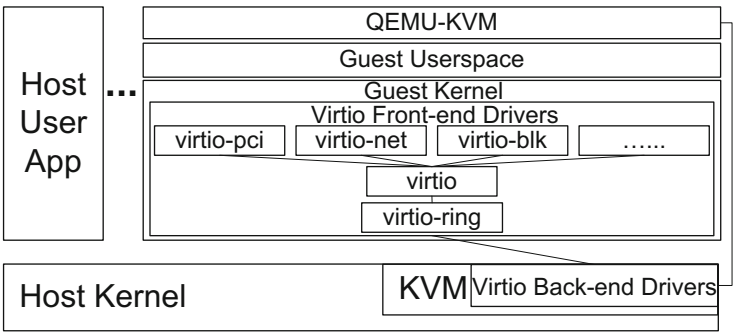

Fig. 2. Virtio architecture

Many virtualization solutions have their own para-virtualization I/O drivers with varying features and optimizations. `virtio`, developed by Rusty, aims to provide a standardized virtual I/O mechanism that works on multiple VMMs. To this end, virtio abstracts a common set of emulated devices and the VMM exports them through a common application binary interface (ABI). In this way, a particular device driver in guest OS can work with multiple VMMs as long as these VMMs implement the required behaviors in their back-ends.

Current virtio implements a virtio-over-PCI model to discovery virtio devices. Each virtio device is driven by two-layered guest-to-VMM communication.

Device Discovery. Because of the universal use of PCI devices, most VMMs support some forms of PCI emulation and most OSes have easy-to-use PCI driver model. A virtio-over-PCI model simplifies the coding of both sides in the virtualization environment.

Any PCI device with vendor ID of 0x1AF4 and Device ID 0x1000 through 0x10FF is recognized as a virtio device. The `probe` function of the PCI driver first allocates necessary resource (port I/O, MMIO regions and interrupt number) for the PCI device, and then calls `register_virtio_device()`, which puts the device on a virtual bus, namely `virtio-bus`. Virtio drivers pick up the devices on `virtio-bus` and recognize the particular `virtio` device based on the subsystem vendor and device ids of its underlying PCI device.

Virtio. The `struct virtio_device` passed in to the virtio driver contains a `virtio_config_ops` struct, which is an array of function pointers. These functions can be used to configure/reset the device by reading/writing a configuration space in the first I/O region of the PCI device. In addition, a dedicated function `find_vq()` can be used to instantiate several `virtqueues` which conceptually attach front-end drivers to back-end drivers.

Transport Abstraction and Virtio_ring. `Virtqueue` is the second abstraction layer for transport. There are functions on it to (1) write new buffers (2) get used buffers (3) notify VMM that new buffers have been added, etc. In addition, when VMM consumes the data and feeds back results to the guest, a callback function is called as an interrupt handler. This callback function is assigned by function `find_vq()` when the `virtqueue` is instantiated.

Theoretically, this layer can be implemented in any way, provided that the guest and the VMM abide by the same rule. Current virtio implements `virtio_ring`, a simple ring-based scheme. The buffers are represented as scatter-gather list, which are chained by the `vring_desc` data structure. Newly added buffers and used buffers are indicated by `available_ring` and `used_ring` respectively. The addresses of the buffers are allocated dynamically inside the guest. To inform the VMM of these addresses, guest writes "guest-physical" addresses to the configure space. The actual "host-virtual" addresses can be calculated by simply adding an offset.

5 Design and Implementation

As shown above, `virtio` exhibits an extreme slim architecture and a flexible interface that greatly reduce the effort to port it to other platforms. Although its original purpose is to standardize virtual I/O devices, we find that it has the building block to implement a virtual cryptographic device. Indeed, `virtqueue` is an ideal place to exchange requests and responses.

In this section, we first explain the design goals of `virtio-ct`. Then we show the way we export keys to the VM. Next, we describe the implementation of actual cryptography service. Finally, the user API and usage are demonstrated.

5.1 Design Goals

The most primary design goal of virtio-ct is that cryptographic keys and sensitive intermediate state during computation should never be accessible by the VM. To this end, the shared buffers that are accessible by both sides should be restricted to those only contain the input/output of the computation. The actual RSA keys and its context should never be exposed to the memory space of the VM.

Meanwhile, every access to the key should be strictly audited and notified. Otherwise, malicious processes could stealthily sign any data it wants to sign once the OS kernel is compromised.

5.2 Key Initialization

`virtio-ct` instantiates two pairs of `virtqueues` during initialization. A pair of `virtqueues` is a channel between VMM and VM: one for transmission and one for receiving. One pair of them serves for the cryptographic computation. The left one is reserved for the purpose of key management, which communicates between VMM and VM about the key information as shown in Fig. 3.

Management Channel. By default, a `virtio-ct` device only contains a management channel and a pair of `virtqueues`. The actual RSA token is specified as a separate device, namely `virtio-ct-token`. A `virtio-ct-token` device is

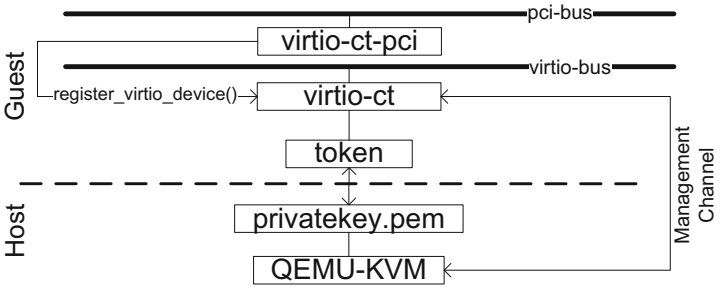

Fig. 3. `virtio-ct` architecture

back-ended by a RSA private key file in PEM format, which is decrypted by a pass phrase when the VM is launched. During initialization of `virtio-ct`, it sends management messages through the reserved management channel to add the token to `virtio-ct` dynamically. In particular, we define the following commands:

- `VIRTIO_CT_READY`: VM notifies VMM that `virtio-ct` is ready for use.
- `VIRTIO_CT_TOKEN_ADD`: VMM sends information about `virtio-ct-token` to the VM.
- `VIRTIO_CT_TOKEN_READY`: VM acknowledges VMM of the added token.
- `VIRTIO_CT_TOKEN_NAME`: VMM sends VM a user friendly string that describes a given token.
- `VIRTIO_CT_TOKEN_PUBKEY`: VMM sends VM the plain-text public key of a given token.

The management message has the following structure (`len` is variable, so this code will not compile):

```
struct virtio_ct_management {
    __u16 cmd;  /* command */
    __u8 buffer[len];  /* command data */
};
```

We do not support hot-plugging and changing RSA keys when VM is running. These features may be useful for VM migration but is not of particular importance in our scenario.

5.3 Cryptographic Service

VM and VMM exchange cryptographic requests/responses though a `virtqueue` pair. VMM calls the OpenSSL cryptographic library to do the actual computation. The request message is sent in the following layout (`in_len` is variable, so this code will not compile):

```
struct virtio_ct_request {
    __u16 cmd;
    __u16 padding;
    __u16 in_len;
    __u8 buffer[in_len];
};
```

`cmd` decides which of the 4 following operations should be performed: (1) encryption with public key (2) decryption with public key (3) encryption with private key (4) decryption with private key. `padding` denotes the padding modes. They are `PKCS1`, `OAEP`, `SSLV23` or `NO-PADDING`. All of them are supported by OpenSSL. We note that public key calculations are completely unnecessary to be pushed in VMM, because public keys are safe to be store in VM. User should consider perform public calculation directly in VM instead of in VMM for efficiency. We

enable CRT, sliding windows and Montgomery multiplication to boost performance, and also RSA-blinding to defeat against timing side channel attack on RSA keys [22].

The response message is much simpler:

```
struct virtio_ct_response {
    __u16 out_len;
    __u8 buffer[out_len];
};
```

If an error occurs (different padding methods have different restrictions on input length), `out_len` is 0 and `buffer` is omitted.

Audit. Although the primary goal of `virtio-ct` is not on access control of the cryptographic service, we do not want unauthorized RSA operations to be performed stealthily without the RSA key owner knowing about it. We write a log whenever VM requests for cryptographic service. Meanwhile, just like a real HSM, `virtio-ct` makes a sound by driving the *pc-speaker* that is a standard component of the PC platform to notify the user. Note that audit is accomplished in the VMM, so the VM cannot suppress this when it issues cryptographic request.

5.4 Use Case

This section first demonstrates the user interface to launch a VM with `virtio-ct` support. Then we show the application programming interface (API) for Linux developers. The implementation of windows driver is in progress.

User Interface. `virtio-ct` consists of a virtual PCI device (`virtio-ct`) and a cryptographic token (`virtio-ct-token`) that is logically attached to it. To add cryptographic token support to a VM, users append a `virtio-ct-pci` and a `virtio-ct-token` option to the QEMU command line. `virtio-ct-pci` is interpreted into a `virtio-ct` virtual device as shown in Fig. 3 while `virtio-ct-token` is the actual token, which requires a PEM formatted private key file as back-end. In Fig. 4, the VM is assigned a cryptographic token that is associated with a distinct printable identifier "key0" thought the `name` argument and an encrypted private key file "/data/prikey0.pem" through the `privatekeypath` argument.

```
File  Edit  View  Search  Terminal  Help
[root@localhost virtio-ct]#  qemu-system-x86_64 -enable-kvm -cpu host -m 4096 -smp 6 \
> ./image.qcow2 -display vnc=:0 \
> -device virtio-ct-pci \
> -device virtio-ct-token,name=key0,privatekeypath=/data/prikey0.pem
Enter PEM pass phrase for /data/prikey0.pem:
```

Fig. 4. `virtio-ct` command line options

API Structure. `virtio-ct` supports two categories of APIs. The `sysfs` attributes are used to export token's name and public key. These are useful for device identification. For example, `udev` rules can be configured to create symlink to the token device by its name (Fig. 5).

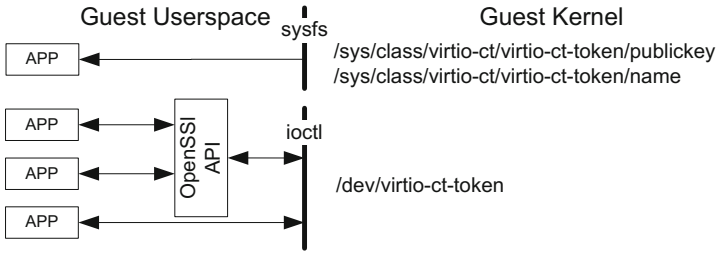

Fig. 5. `virtio-ct` API

The other category of API is used for the actual cryptographic services described in Sect. 5.3. Accordingly, there are 4 kinds of `ioctl` commands to perform a public/private encryption/decryption. The used padding mode is included in the `ioctl` messages. Furthermore, we encapsulate this device specific API to a universal applicable module – an OpenSSL engine. When the RSA key is loaded through the function `ENGINE_load_private_key()`, the corresponding virtual token can be used to do cryptographic computation by calling widely used OpenSSL routines. This would be useful for the easy integration of `virtio-ct` into other cryptographic programs.

6 Evaluation

6.1 Performance

We show the throughput of the `virtio-ct` prototype, and compare it with the native OpenSSL implementation inside the VM. In addition, we measured the system load of both the VMM and VM when calling `virtio-ct` service. All the experiments were conducted using 2048-bit RSA keys to do a private key encryption, and the padding mode is `PKCS1`. The target machine is a Lenovo PC with an Intel core i7-4770S CPU and 8 GB memory. The used CPU has 4 physical cores with hyper-threading support.

Throughput was measured in multi-core mode. Specifically, we enabled hyper-threading on the host and assigned 6 vCPUs and 4 GB memory to the VM. Each thread requests for private key encryption in an infinite loop. In Fig. 6, we can see that the throughput of `virtio-ct` does not grow as the concurrency level increases, instead, it decreases slightly. This is because only one I/O thread can execute in the VMM and the increased threads only adds the burden of

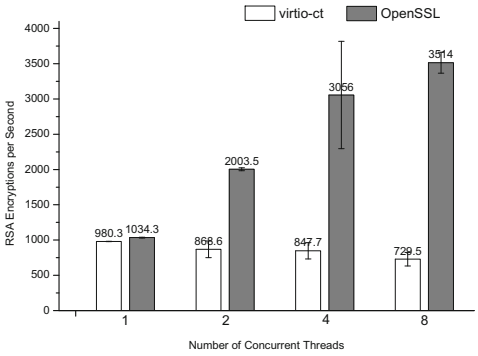

Fig. 6. Throughput

task scheduling. We will expand this in the next section. One interesting observation is that the standard deviations of both solution are extraordinarily high when there are more threads. We attribute it to hyper-threading. In fact, when 2 threads are running in the same physical core, the speed is degraded as these two logical threads share the same set of execution unit.

We next measured the latency for each cryptographic request. To minimize the effect of hyper-threading and task scheduling, we assigned a single thread to the VM and disabled hyper-threading on host. The average latencies are 1041.3 ms and 964.9 ms for `virtio-ct` and OpensSSL respectively. These values are very close to the average processing time drawn in Fig. 6. This is somehow expected: `virtio-ct` works in blocking mode and the RSA computation time is much more than the time spent on context switching.

We then used the `top` utility to record the CPU load of both the host and guest machine when there is one `virtio-ct` thread. Obviously, one thread achieves the maximum throughput. System loads were recorded per second for 1000 s, and the statistics on average were calculated as shown in Table 1. Apart from idle state, VM spent most of time in kernel mode and on handling hardware interrupts. Note that VMM notifies VM of the returned data by virtual-interrupt injection. In contrast, VMM spent most of time in user space. Indeed, RSA computation runs inside the QEMU process which is in user mode.

Table 1. CPU state percentages for VM and VMM. The host machine has 4 physical cores with hyper-threading enabled. VM is attributed 6 vCPUs and launches a single thread to call `virtio-ct` service in an infinite loop.

	User*	Kernel	Idle	Wait for I/O	Hardware interrupts	Software interrupts
VM	0.043	13.830	74.334	2.372	8.819	0.602
VMM	24.987	1.142	73.735	0.098	0.032	0.006

*User state presents time running both niced and un-niced user processes.

Efficiency Issues. As shown above, the maximum throughput of `virtio-ct` is close to that of native OpenSSL library with one vCPU, regardless of the attributed number of vCPUs. This indicts that the current QEMU implementation can only execute one I/O thread simultaneously. Figures 7 and 8 list some code snaps that handle I/O events. I/O could be served either in a dedicated QEMU I/O thread or in vCPUs threads. However, there is a global mutex that synchronizes core QEMU code across them. That is, only one thread can execute code that handles I/O that may operate on global structures of QEMU. Note that when there is not much I/O events, vCPU is in the guest mode for most of the time, so several vCPUs can runs in parallel for computation-intensive programs.

The current QEMU thread structure is sufficient for real I/O tasks when most of CPU time are waste waiting for I/O completion. For virtual I/O that does not involve real peripheral, like `virtio-ct`, the constraint that all the I/O must be serialized is a big hit for performance. One of our future work is to allocate more threads that are isolated from the QEMU context, so that cryptographic service can be executed in parallel in these extra threads. In fact, VNC [23] and SPICE [24] display protocols which involve intensive computation (video codecs, display encryption, etc.) have adopted similar solutions.

6.2 Security

We performed extensive tests that observe the memory space of the VM to ensure that there is no occurrence of private RSA keys. Inside the QEMU console, we used the `dump-guest-memory` command to dump the memory image of the VM and the `info registers` command to obtain register contents, and then used various ways to find RSA keys. We first invoked an automatic tool called RSAKeyFinder [25]. It searches for the patterns of DER-encoded RSA keys to find suspicious memory contents. We successfully find out some occurrences of RSA keys, but none of them is that used in `virtio-ct`. On the contrary, when we ran the tool on the memory dump of the QEMU process, all the used keys were recovered.

```
int main_loop_wait(...)
{
    ......
    qemu_mutex_unlock_iothread();
    g_poll_ret = qemu_poll_ns(
        poll_fds, ...);
    qemu_mutex_lock_iothread();
    if(g_poll_ret > 0)
    /* process I/O */
}
```

Fig. 7. Dedicated I/O thread

```
int kvm_cpu_exec(CPUState *cpu)
{
    ......
    qemu_mutex_unlock_iothread();
    run_ret = kvm_vcpu_ioctl(cpu,
        KVM_RUN, 0);
    qemu_mutex_lock_iothread();
    kvm_arch_post_run(cpu, run);
    /* process I/O */
}
```

Fig. 8. vCPU thread

The second method is using a simple binary matching program `bgrep`. As we know the plain-text of the keys, we used `bgrep` to match the key string (including p, q, d and other CRT elements). It turned out that we never found a binary sequence that overlaps for more than 3 bytes with any key. These experiments proves that there will be no occurrence of escrowed RSA key copies when employing `virtio-ct`. The compromise of guest OS will not effect the secure storage of keys, provided that the VMM is implemented correctly.

7 Conclusions and Future Work

We present `virtio-ct`, a virtual cryptographic token in the KVM virtualization environment. `virtio-ct` assembles a real HSM in that it never exposes the real keys to the guest OS, so that the compromise of the guest OS will not threaten the secrecy of the keys. Moreover, audit is enforced in a mandatory way when the `virtio-ct` service is called. Because `virtio-ct` is a software solution and most personal computers have support for hardware virtualization, it is more flexible and economical to achieve cryptographic key isolation, compared with HSM solutions. Our prototype achieves nearly 1,000 times RSA private operations per second on a mainstream Intel desktop processor.

Future Work. We intend to extend the prototype with the following features.

1. Dedicated cryptographic threads to boost performance.
2. Physical memory attack resistance: We are resorting to solutions such as PRIME [26] and Copker [27] to add cold-boot resistent in the VMM. As a result, `virtio-ct` achieves comparable security strength with HSM in all aspects.
3. Conformation to the PKCS#11 standard [28]. We plan to support more cryptographic algorithms through this widely used standard API.

Acknowledgments. The authors would like to thank the anonymous reviewers for their helpful suggestions and valuable comments. Le Guan, Jiwu Jing, Jing Wang and Ziqiang Ma were partially supported by National 973 Program of China under award No. 2014CB340603. Fengjun Li was partially supported by NSF under Award No. EPS0903806 and matching support from the State of Kansas through the Kansas Board of Regents, and the University of Kansas Research Investment Council Strategic Initiative Grant (INS0073037).

References

1. Chow, J., Pfaff, B., Garfinkel, T., Rosenblum, M.: Shredding your garbage: reducing data lifetime through secure deallocation. In: 14th USENIX Security Symposium (2005)
2. The MITRE Corporation, CWE-226: Sensitive information uncleared before release (2013). https://cwe.mitre.org/data/definitions/226.html
3. The MITRE Corporation, CWE-212: Improper cross-boundary removal of sensitive data (2013). https://cwe.mitre.org/data/definitions/212.html

4. Tang, Y., Ames, P., Bhamidipati, S., Bijlani, A., Geambasu, R., Sarda, N.: Cleanos: Limiting mobile data exposure with idle eviction. In: 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12), Hollywood, CA, pp. 77–91 (2012)
5. National Vulnerability Database, CVE-2014-0160. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-0160
6. Engler, D., Chen, D., Hallem, S., Chou, A., Chelf, B.: Bugs as deviant behavior: a general approach to inferring errors in systems code. In: 18th ACM Symposium on Operating Systems Principles, pp. 57–72 (2001)
7. Lafon, M., Francoise, R.: CAN-2005-0400: Information leak in the Linux kernel ext2 implementation (2005). http://www.securiteam.com
8. Guninski, G.: Linux kernel 2.6 fun, Windoze is a joke (2005). http://www.guninski.com
9. Desmedt, Y.G., Frankel, Y.: Threshold cryptosystems. In: Brassard, G. (ed.) CRYPTO 1989. LNCS, vol. 435, pp. 307–315. Springer, Heidelberg (1990)
10. Itkis, G., Reyzin, L.: SiBIR: signer-base intrusion-resilient signatures. In: Yung, M. (ed.) CRYPTO 2002. LNCS, vol. 2442, pp. 499–514. Springer, Heidelberg (2002)
11. Parker, T.P., Xu, S.: A method for safekeeping cryptographic keys from memory disclosure attacks. In: Chen, L., Yung, M. (eds.) INTRUST 2009. LNCS, vol. 6163, pp. 39–59. Springer, Heidelberg (2010)
12. Mavrogiannopoulos, N., Trmač, M., Preneel, B.: A linux kernel cryptographic framework: decoupling cryptographic keys from applications. In: Proceedings of the 27th Annual ACM Symposium on Applied Computing, ser. SAC 2012, pp. 1435–1442 (2012)
13. Safe Net, Luna g5 usb-attached hsm. http://www.safenet-inc.com/data-encryption/hardware-security-modules-hsms/luna-hsms-key-management/luna-G5-usb-attached-hsm/. Accessed July 2014
14. Kernel Based Virtual Machine. http://www.linux-kvm.org/page/Main_Page
15. QEMU open source processor emulator. http://wiki.qemu.org/Main_Page
16. Russell, R.: Virtio: towards a De-facto standard for virtual I/O devices. SIGOPS Oper. Syst. Rev. **42**(5), 95–103 (2008)
17. Halderman, J., Schoen, S., Heninger, N., Clarkson, W., Paul, W., Calandrino, J., Feldman, A., Appelbaum, J., Felten, E.: Lest we remember: cold boot attacks on encryption keys. In: 17th USENIX Security Symposium, pp. 45–60 (2008)
18. Berger, S., Cáceres, R., Goldman, K.A., Perez, R., Sailer, R., van Doorn, L.: vTPM: virtualizing the trusted platform module. In: 15th USENIX Security Symposium, vol. 15 (2006)
19. Intel Corporation, Intel software guard extensions. https://software.intel.com/en-us/intel-isa-extensions#pid-19539-1495. Accessed July 2014
20. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the art of virtualization. SIGOPS Oper. Syst. Rev. **37**(5), 164–177 (2003)
21. Shinagawa, T., Eiraku, H., Tanimoto, K., Omote, K., Hasegawa, S., Horie, T., Hirano, M., Kourai, K., Oyama, Y., Kawai, E., Kono, K., Chiba, S., Shinjo, Y., Kato, K.: Bitvisor: a thin hypervisor for enforcing i/o device security. In: Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, pp. 121–130 (2009)
22. Brumley, D., Boneh, D.: Remote timing attacks are practical. Comput. Netw. **48**(5), 701–716 (2005)
23. Virtual Network Computing. http://www.realvnc.com/

24. SPICE: Simple Protocol for Indenpendent Enviroment. http://www.spice-space.org/
25. Heninger, N., Feldman, A.: RSAKeyFinder. https://citp.princeton.edu/research/memory/code/
26. Garmany, B., Müller, T.: PRIME: private RSA infrastructure for memory-less encryption. In: 29th Annual Computer Security Applications Conference (2013)
27. Guan, L., Lin, J., Luo, B., Jing, J.: Copker: Computing with private keys without RAM. In: 21st ISOC Network and Distributed System Security Symposium (2014)
28. RSA Laboratories, PKCS#11: Cryptographic Token Interface Standard. http://www.emc.com/emc-plus/rsa-labs/standards-initiatives/pkcs-11-cryptographic-token-interface-standard.htm