Utilizing GPU Virtualization to Protect the Private Keys of GPU Cryptographic Computation

Ziyang Wang[1,2,3], Fangyu Zheng[1,2(✉)], Jingqiang Lin[1,2,3], and Jiankuo Dong[1,2,3]

[1] Data Assurance and Communication Security Research Center, CAS, Beijing 100093, China
{wangziyang,zhengfangyu,linjingqiang,dongjiankuo}@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

Abstract. Nowadays graphics processing units (GPUs) have become popular parallel computing platforms known as General-Purpose GPU (GPGPU) computing. GPUs thereby are chosen by some security researchers as cryptographic accelerators to secure massive volumes of transactions. However, their security issues are ignored in spite of their popularity and performance. There are some possible information leakages faced with malicious attacks or even in the normal GPU computing. Our objective is to secure the confidentiality of cryptographic keys in GPU computing environments and provide easy-to-use programming with few constraints. In this paper, we propose a prototype in Linux, a system of GPGPU computing solution empowered by GPU virtualization technology, which keeps encrypted keys in guest machine to protect secret keys from leakage even in the event of full system compromise. With the API interception and redirection of CUDA, applications in Virtual Machines (VMs) can access the GPU device in a transparent way. Besides, we use `virtio`, a dedicated virtual I/O device, to transfer data between virtual and host machines in high performance. In our current study, we evaluate our prototype with the GPU implementation of ECC. We show that it can protect private keys of GPU cryptographic computation and it also incurs low performance penalty compared with the native environment, therefore demonstrating the prototype is secure and requires reasonable overhead.

Keywords: GPU · GPGPU · GPU virtualization
Information leakage · Isolation

This work was supported by National Natural Science Foundation of China under Award no. 61772518.

D. Naccache et al. (Eds.): ICICS 2018, LNCS 11149, pp. 142–157, 2018.
https://doi.org/10.1007/978-3-030-01950-1_9

1 Introduction

Over the last decade, graphics processing units (GPUs) have been increasing used both as accelerating graphics rendering engines and parallel programmable processors due to their high computing power and low price.

With hundreds to thousands of streaming processing cores, modern GPUs is to speed up computations in the single-instruction-multiple-data (SIMD) fashion, providing ample computation cycles and high memory bandwidth to massively parallel applications. As a result, GPUs have quickly been applied in a broad spectrum of applications.

Meanwhile, the expanding demand of cryptographic operations for secure communication and authentication requires high-performance implementations. In fact, the GPU-based implementations of cryptographic operations (e.g., RSA, AES, ECDSA, SHA-1) achieve significantly higher throughput and efficiency than CPU implementations [9,11,12,23]. GPUs are leveraged to offload cryptographic workloads from CPUs. For example, the GPU implementation of AES achieves up to 28x higher throughput.

Although the GPU-based implementations of cryptographic operation aims at security, a thorough analysis of the GPU environment has not been well studied. As regards security and isolation, they are not considered as important as performance. In fact, GPU and CPU architecture are quite different, therefore they face different threats.

For discrete GPUs' architecture, their independent memory and computational resources are physically partitioned from CPU, which seems to make it plausible that GPUs could be used as *secure co-processors*. In CCS '14, Vasiliadis et al. [29] proposes PixelVault, which is a system implementing AES and RSA for keeping sensitive information (including cryptographic keys) and performing cryptographic operations exclusively on GPU. PixelVault chooses GPU registers, which are reported to be automatically reset to zero when the kernel is loaded, as the secret and private keys storage. Intermediate sensitive data is kept encrypted by master key in GPU global memory. No doubt the master key is stored in GPU registers. Any computation with the secret keys is exclusively limited to those registers. As a result, PixelVault prevents even privileged host code from accessing any sensitive code or data on GPU. By exposing private keys in plaintext only in GPU registers, and keeping PixelVault's critical code exclusively in the GPU instruction cache, PixelVault seems to be a promising approach to prevent even privileged host code from accessing any sensitive code or data on GPU.

With a different technology roadmap, in 2016, Kim et al. [13] propose OBMI which is a SMM-based (System Management Mode) framework for bootstrapping secure cryptographic operations on GPGPUs while preserving robustness, efficiency and programmability. Unlike PixelVault, they store keys in GPU cache which also cannot be accessed by CPU processes and cannot be accessed after termination of GPU kernel. However, data in GPU cache is beyond control of programmer, it is critical to ensure data remain in the cache and can not be evicted. To subvert this issue, they store the key in the constant cache using

SMM before system booting. They also utilize SMM for isolating authenticated GPU kernel in instruction cache. Thus, only SMI (System Management Interrupt) and trusted kernel can access the key. By handling sensitive data only in SMM, OBMI can secure cryptographic operations.

Unfortunately, while some characteristics of GPU architecture and execution model are officially confirmed, some are poorly documented and not validated experimentally. Indeed, Zhu et at. [32] demonstrate how unpublished or recently introduced features of GPUs may bypass the protection mechanisms of PixelVault and compromise the whole system. They refute the following security assumptions of PixelVault in details. Exploiting memory mapped input output (MMIO) registers, they can invalidate the GPU instruction caches and replace them with their own malicious code from running kernels. Using recent changes in debugging support, privileged users are able to attach any running kernel and read the GPU registers, effectively extracting the secret keys. What's worse, it is unclear how to disable this capability. And for OBMI, becasuse However as mentioned before [32], unpublished MMIO registers are able to flush the instruction cache, allowing to inject malicious code. Consequently, it breaks the security assumption of both PixelVault and OBMI.

As a summary, any kind of information leakage from security-sensitive applications (e.g., those runs security protocols or cryptographic algorithms) would severely undermine the trustworthiness of GPU computing. Thus in order to protect the secret keys under a range of memory leakages and threats to the underlying system, we propose a key-protection isolation mechanism on GPGPUs with system-level virtualization technology. The contributions of this paper are threefold:

1. We propose a secure GPU computing model for the GPU-based cryptographic service. We sugguest a GPU virtualization approach for cryptographic computations by API remoting method, which isolates master key and keeps accelerating operations safe. To the best of our knowledge, we are the first to utilize GPU virtualization technology to solve the security issues of GPU.
2. Based on the proposed model, we implement a prototype on the commodity GPU with QEMU-KVM with various kinds of optimization. The extra virtualization layer also allows us to introduce mechanisms for checking the integrity of the accelerated GPU code, which is previously very complex to implement in GPUs.
3. By comparing the native throughput of ECC with GPU virtualization through various experiments, we evaluate the performance overhead of our solutions. The evaluations show that our approach incurred only a negligible performance degradation with preventing private keys from leakage of GPU cryptographic computation.

The rest of the paper is organized as follows: we begin by providing necessary background for the current work. Then we describe our design in Sect. 3. In Sect. 4, the detailed implementation is revealed and we evaluate the performance. We discuss security analysis in Sect. 5. Finally, we conclude the paper in Sect. 6.

2 Preliminaries and Related Work

This section gives some basic introduction to GPU, GPU virtualization and some attacks and defenses on cryptographic keys in modern processors.

2.1 GPU Basis

The computational capabilities of GPUs for executing parallel applications are based on hundreds or thousands of processing cores and a high bandwidth memory architecture. A GPU has several Streaming Multiprocessors (SM) which are in turn composed of Streaming Processor cores (SP, or CUDA cores), registers, caches.

A GPU application consists the host code running on the CPU and kernels which runs on the GPU. GPU kernels are special functions executing n times in parallel by n different threads. The number of threads can be specified at kernel launch time.

Running a task on GPU follows three steps:

- The DMA controller transfers the input data from host memory to GPU memory.
- The host application launches the kernel which runs on GPUs.
- The DMA controller transfers the output data from GPU memory to host memory.

As CUDA is becoming a prevalent programming model of GPGPU, we focus on CUDA runtime API while virtualizating GPUs.

2.2 GPU Virtualization

Although virtualization provides a wide range of benefits, such as system security, ease of management, isolation, and live migration, virtualizing GPUs is a relatively new area of study and remains a challenging endeavor which is due to undisclosed details of GPU implementation and unstandardized GPU architectures.

API remoting approach is a kind of protocol redirection, which virtualizes GPUs in a simple way and without singnificant performance penalty by providing a GPU wrapper library to a guest OS to intercept GPU runtime calls. This approach does not adopt custom GPU driver in the guest [10]. vCUDA [25] and rCUDA [3] are recent projects using API remoting in GPU virtualization.

vCUDA provides a CUDA wrapper library and virtual GPUs (vGPUs) in the guest and the vCUDA stub in the host. vGPUs are created per application by the wrapper library and give a complete view of the underlying GPUs to applications. Instead of emulation, rCUDA creates virtual CUDA-compatible devices on machines without GPU by adopting remote GPU-based acceleration.

However, these frameworks either rely on the scheduling mechanisms provided by the CUDA runtime, or allow applications to execute on GPU in sequence, possibly leading to low resource utilization and consequent suboptimal performance.

2.3 Attack and Defense on Cryptographic Keys in CPU

As long as the secrecy of cryptographic keys involved in cryptographic operations is guaranteed, the cryptosystem is trustworthy even if it has been compromised. Nevertheless keeping cryptographic keys safe is still a great challenge in any cryptosystem [16]. During cryptographic operations, private keys are always loaded into main memory as plaintext, therefore private keys are prone to memory disclosure attacks.

A malicious program can exploit Meltdown [17] and Spectre [14] in modern processors to steal data from the main memory, dumping passwords, personal photo, emails, instant messages and so on.

Although various mechanisms have been proposed in memory protection [28,30], the main memory is still vulnerable to physical attacks, such as cold-boot [8], DMA attack [1,27], which could bypass the protections of OS. To prevent cold-boot attack, AESSE [21], TRESOR [22], and Amnesia [26] store AES keys exclusively in CPU registers. PRIME [5] and Copker [6] implements the RSA algorithm in AVX registers and cache respectively. While Mimosa [7] utilizes hardware transactional memory to protect the RSA cryptographic operations from cold-boot and memory disclosure attacks.

However the CPU-bound encryption approach requires the integrity of the OS kernel. Any compromised OS kernel can easily leak the register or the cache within the CPU. TRESOR-HUNT [1] exploits DMA to inject malicious code into the OS kernel memory and then access the keys in registers.

2.4 Attack and Defense on Cryptographic Keys in GPU

Not only for CPUs, recent works have started investigating the security vulnerabilities of GPUs. Some works notice the GPU driver does not erase memory after kernel termination, indicating that they can leak sensitive data [24,31].

Memory isolation policies enforced by a CPU cannot be applied to GPU memory automatically, so any discrepancy between CPUs and GPUs can lead to unexpected information leakages. By exploiting such vulnerabilities, Pietro et al. [24] recover both plaintext and encryption key of AES from GPU global memory. For thwarting information leakage in GPUs, it is believed that the best solution is memory isolation enhancements performed at the driver/hardware level. Nevertheless, it would be better that CUDA should allow OS to monitor usage and control access to GPU resources.

As modern GPUs share virtual and even physical memory with CPUs, buffer overflows becomes possible in GPU and can lead to remote code execution, corruption on sensitive data and security problems as CPU-based overflows [2, 20]. Erb et al. [4] present a tool that utilizes canaries to detect buffer overflows caused by GPGPUs kernel in OpenCL GPU applications.

A recent study shows that remanent data in GPU memory can be retrieved since GPU does not automatically zero its memory after termination. [15,24,31] Even implementing an appropriate erasing operations for the GPU memory, attackers with GPU driver privileges can also access GPU memory with MMIO

registers [19]. Some researchers treat discrete GPUs as secure co-processors storing private keys in GPU registers [29] or GPU cache [13] while offloading cryptographic operations onto GPUs. Unfortunately, due to the widespread commercial strategy to hide implementation details from competitors, manufacturers of GPUs are reluctant on publishing the internals of their solutions [18], which implies the discrete GPUs cannot be trusted as secure co-processors, rather, they may host stealthy malware [32].

3 System Design

3.1 Threat Model and Design Goals

Threat Model. We intend to provide a isolated cryptographic computation environment using GPU in virtual machine from the OS in commodity platform. In this situation, the objective of the adversary is to leak the sensitive information of cryptographic operations from victim's system. We assume that the adversary has the ability to fully control over the VM and obtain root privilege through intrusion attacks. We consider the malicious user has no physical access to the computer. Otherwise the victim machine is venerable to hardware-based attacks such as cold-boot attack or DMA attack.

The underlying VMM is mostly safe so that even if the OS of VM is compromised, the hacker cannot escape the guest virtual machine and execute code on hypervisor or host operating system. Moreover, we ignore denial-of-service attacks.

Design Goals. Our most primary goal is to design a safe environment for GPU accelerating cryptographic computation without leaking secret keys. This implies that no keys or sensitive information should get into memory. Considering this policy, we can isolate the GPU and OS with a master key from a virtual machine. To restrain cryptographic operations from dealing with secret keys, they are transferred from VM to host using secure channel. The actual secret keys and related sensitive information should never be exposed to the memory of VM. In that case even if the VM is compromised, no sensitive information in VM memory can be leaked.

Meanwhile, we need to guarantee the throughput of cryptographic computations. The performance of cryptographic operations should be not effected obviously.

3.2 System Architecture

The framework we propose is organized in three main architectural features. By using CUDA API Remoting, we implement GPU virtualization in guest OS. CUDA cryptographic applications can utilize GPU with original API functions in the same manner as a typical GPGPU program. However, secret keys must

not be transferred into memory of guest OS in case they are disclosed via malicious attacks. All we suggest to manage keys securely is to isolate the master key in VMM, while secret keys from any application need to encrypt/decrypt via the master key. Moreover, to prevent leakage of sensitive information from GPU memory, we also verify the validity of the CUDA fat binary before application launches a request. If there is no information of CUDA fat binary in our white list, the system denies the request and records this abnormal behavior. In this way, users can launch secure cryptographic operations on virtualized environment. The architecture will be discussed in depth in the following sections.

Isolation Using GPU Virtualization. The first step of our isolation mechanism for securing accelerating cryptographic computation is the ability to utilize GPU in virtual machine. As we all know, GPU vendors somehow are not willing to provide general purpose GPU virtualization solution. They tend not to publish the source code and implementation details of their GPU drivers, which are essential for virtualizing GPUs at the driver level, for commercial reasons. Even when driver implementations are unveiled, for example, by reverse engineering methods, significant changes are introduced with every new generation of GPUs to improve performance. As a result, specifications revealed by reverse engineering become useless.

In a word, there are no standrad interfaces for virtualizing GPU devices in driver level.

Fortunately the API remoting approach overcomes aforementioned limitations, because it can emulate a GPU execution environment without exposing physical GPU devices in the guest OS. The premise of API remoting is to provide guest OS with a custom library which contains the same API as the original library. However the library intercepts GPU calls from the application and redirects the request with proper parameters to the host OS for execution as normal calls through shared memory or sockets. Only results are delivered to the application through wrapper library in reverse.

It is flexible that the wrapper library can be dynamically linked to existing applications at runtime. What is more important is that API remoting approach incurs negligible performance overhead. In addition, this approach can be monitored by underlying hypervisors as the virtualization layer is implemented in user space.

Key-Protection Mechanism. The main focus of our work is the key-protection mechanism during the GPU accelerating cryptographic operations. Instead of storing the master key in registers or cache of GPU like PixelVault and OBMI do, we decide to keep master key in VMM. All keys residing in VMs are encrypted by the master key. Thus only during cryptographic operations, VMM manages to decrypt cryptographic keys with the master key and uploads the actual keys into GPU global memory where kernel can retrieve. Therefore, even if adversaries manage to acquire the secret keys from the memory space of VMs, they would get encrypted content which is unuseful.

GPU Binary Verification. Although our key-protection mechanism can securely upload the secret key into the GPU global memory, it is useless if the GPU kernel is compromised. Thus adversary can launch a malicious GPU code-injection attack and patiently leak the key from the GPU global memory. In order to prevent such attack, verifying the integrity of GPU kernel is a prerequisite for protecting secret keys. Before launching the accelerating cryptographic operation in VM, VMM checks the integrity of the GPU kernel. After validating authenticated GPU code, VMM allows the GPU binary to execute then. Otherwise, the cryptographic operations are rejected and recorded in abnormal logs. With the proposed method, we can securely execute arbitrary, authenticated GPU code without any tampering.

4 Implementation and Performance Evaluation

Based on the design principles, we describe various aspects of our prototype in detail, and propose a general system architecture and some more details in the implementation of our prototype.

4.1 Architecture Overview

The overall design and process details of the proposed solution are shown in Fig. 1. In this architecture, we choose `virtio`, a standard for para-virtualization I/O devices, as the transfer channel, which we discuss in following sections. By using API remoting approach, CUDA driver is not essential for guest OS.

First, when a CUDA application demands a GPU service, it can dynamically link to wrapper library and invoke CUDA runtime API as the typical GPGPU program. Wrapper library intercepts calls before the calls reach the GPU driver in the guest OS and redirects them to virtio front-end driver through ioctl. Then guest driver forwards API requests with proper parameters via virio buffer to back-end driver in VMM. If it is the first time running for application, VMM verifies the fat binary file using HMAC and then checks it whether in our white list. Only passing the validation, API requests can be executed as the same

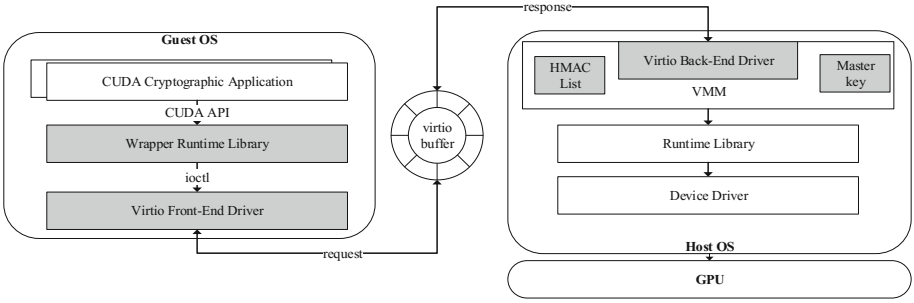

Fig. 1. The architecture of the proposed prototype.

real runtime APIs do. Otherwise, VMM denies of execution. Finally, the results should be sent back to the application in reverse path.

4.2 Implementation Details

API Remoting GPU Virtualization. In general, the typical phase for execution of a kernel requires several steps, which we illustrate using the vector addition as an example:

1. **Initialization.** The process obtains the GPU module from the CUDA binary, which comprises the CUDA fat binary code and other related data such as statically allocated variables.
2. **Memory Allocation.** The process requests memory allocation on the GPU for the data used by kernel execution.
3. **Input Data Transfer.** All the data used by kernel execution must be copied from RAM to GPU global memory allocated in second step.
4. **Kernel Execution.** The GPU code is executed with the parameters and configurations, such as block size and thread size.
5. **Output Data Transfer.** Once the kernel execution is completed, the results in GPU should be transferred to RAM.
6. **Memory Release.** GPU memory which is allocated before is released.
7. **Finalization.** The process releases all the associated resources and quits.

According to our experiments, all the cardinal running APIs we need to complete a typical cuda application is shown in Table 1.

Table 1. The functionality of primary runtime API

Operation	Functionality	Stage
`__cudaRegisterFunction`	Get the handle to kernel called with binary code and function name	Initialization
`__cudaUnregisterFatBinary`	Release the fat binary	Finalization
`__cudaRegisterFatBinary`	Get the handle to the fat binary	Initialization
`cudaMalloc`	Allocate memory on the device	Memory allocation
`cudaConfigureCall`	Configure a device launch	Kernel execution
`cudaSetupArgument`	Configure setup augments	Kernel execution
`cudaLaunch`	Launch a kernel	Kernel execution
`cudaFree`	Free memory on the device	Memory release
`cudaMemcpy`	Copy data between host and device	Input&Output data transfer

The first three functions with "__" prefix are not meant to be called directly by user code but they are so important that `nvcc` compiler injects them into the source code. Their declaration in */usr/local/cuda/include/crt/host_runtime.h*

shows us the interface. While the other functions without underline prefix are directly called by user code and they are declared in */usr/local/cuda/include/cuda_runtime_api.h.*

Each wrapper runtime API invokes `ioctl()` system call for sending requests and getting responses from virtual GPU. In this case, `ioctl()` takes the file descriptor of virtual character device as first argument. The second argument is a dedicated device-dependent request code for each runtiem API. The third argument is an untyped pointer to the request meta structure which we fill with proper parameters.

Data Transfer Between VMs and VMM. Virtio is a de-facto standard for para-virtualization I/O devices and aims to improve performance of accessing devices on guest OSes over the traditional emulated devices. Virtio abstracts a common set of emulated devices which the VMM exports to the VM via normal PCI devices. To boost the I/O performance, a custom device driver in guest OS communicates with the associated back-end service in VMM. Guest OS writes "guest-physical" addresses to the configure space to inform the VMM of buffer addresses. By simply adding an offset the actual "host-virtual" addresses can be calculated in VMM.

Our proposed implementation, virtio-vgpu, consists of a virtual PCI device (virtio-vgpu-pci) and a token (virtio-vgpu-token) that is logically attached to it. To support virtio-vgpu, "`virtio-vgpu-pci`" and "`virtio-vgpu-token`" options should be appended to the QEMU command line when the VM launches. Virtio-vgpu-pci is interpreted into virtual device for guest OS, while virtio-vgpu-token also requires a PEM formatted private key file which stores master key as back-end with the "`keypath`" argument.

Virtio-vgpu provides a front-end driver in guest OS for forwarding requests and returning the response of CUDA runtime APIs. In more detail, the driver specifies ioctl commands for wrapper runtime API calling. All the parameters should be rearranged in buffer with the meta structure and auxiliary data. As the driver is complemented in kernel space, transferred data from user space should be copied to the kernel space. New kernel memory is allocated, filled with content from the user space and then concatenated to the end of buffer. The final buffer is sent to VMM via virtio buffers. The returned buffer is also arranged like this, which contains of meta structure and complementary data. Finally, all essential results are copied to user space.

Besides the front-end driver, the other back-end driver in VMM is responsible for receiving requests, keeping states of objects, executing operations and sending back the responses. The transferred data is analyzed via the information of meta structure located in the front of buffer. As we mention before, VMM responses differently according to dedicated request code from the meta structure. By different request code, VMM calls actual runtime API and forwards back the buffer with the returned value and essential parameters. In addition, VMM also initializes the virtual objects list after validating the authentication of GPU code when dealing with `__cudaRegisterFatBinary` requests. The virtual objects list

manages all pointers to allocated memory, kernel configuration and parameters, events, streams. Still, the device information is collected as well in order to response quickly for the device management, such as `cudaGetDevice` request.

GPU Kernel Authentication. In order to prevent the maliciously modified GPU code compromising the device, our system only allows authenticated GPU code to execute on device. To validate the integrity of GPU kernel, the mechanism is divided into two parts. Firstly, the administrator should obtain the signature of fat binary file from CUDA binary using HMAC-SHA256 signatures in advance. Now the secret key of HMAC is generated from master key. Then those signatures are needed to be converted to base64 for displaying. The fat binary file can be generated from the source code by `nvcc` compiler with the option "`--fatbin`". Meanwhile VMM maintains a white list to dynamically manage base64 strings of HMAC signatures of GPU code. The white list is implemented as a file appended to **virtio-vgpu-token**.

```
typedef struct {
        int magic;
        int version;
        const unsigned long long* data;
        void *filename_or_fatbins;  /* version 1: offline filename,
                * version 2: array of prelinked fatbins */
} __fatBinC_Wrapper_t;
```

Fig. 2. Fatbin control structure.

Secondly, what confronts us is how to extract the fatbin file from the CUDA binary file using runtime APIs. Luckily, we can utilize some control structure information from CUDA include directory. `__cudaRegisterFatBinary` takes the pointer to fatbin control structure as input, the structure defined in *fatBinaryCtl.h* is shown in Fig. 2. The address of fat binary can be followed by field `data`. Furthermore, the size of fat binary file is controlled by fat binary header structure defined in *fatbinary.h*. Finally, with the address and size, the fat binary file surely is able to be extracted and be transferred to VMM via virtio.

VMM checks the integrity of fat binary file once receiving it from guest OS. By checking whether base64 string converted from HMAC-SHA256 signature of fat binary file exists in while list or not, VMM determines that the fat binary file is authenticated. If the fat binary file is valid, VMM initializes the virtual GPU environment and deal with following requests. Otherwise, VMM rejects the request and reports this abnormal behavior to administrators.

4.3 Performance Evaluation

We assess the performance of the GPU virtualization prototype in comparison to the native machine running on commodity CPU and GPU. Our host OS consists

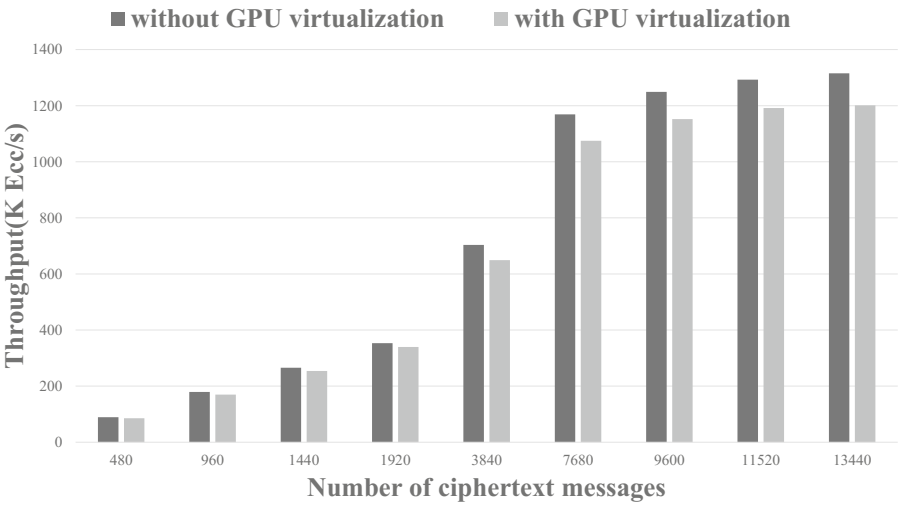

Fig. 3. Performance comparison for Curve25519 with different numbers of requests.

Ubuntu 16.04 x86-64 (kernel v4.4.0), QEMU v1.7.1 and NVIDIA Geforce GTX TITAN BLACK, and the guest OS is CentOS 6.6 (kernel v3.13.7). We implement the wrapper library of CUDA v8.0. To adopt our mechanism for common GPU cryptographic operations, we develop auxiliary runtime API in wrapper library.

The throughput is evaluated via Curve25519 and AES-128 algorithms. Curve25519 is an elliptic curve which is intended to operate at the 128-bit security level. We implement scalar multiplication on Curve25519 by using Montgomery ladder in a constant time. By making full use of the PTX ISA instruction supported by NVIDIA GPUs and making optimization from two aspects, the finite field arithmetic and the curve algorithm, the performance of Curve25519 scalar multiplication has been promoted. For AES, we modify the OpenSSL AES with a 128-bit symmetric key to GPU implementation.

We conduct the experiment to measure the latency incurred by the API remoting approach. In order to measure the impact of our GPU virtualization mechanism on real scene of cryptographic computing, the time imposed by data copying is included from the total time. Figure 3 shows the throughput changes of Curve25519 with and without our proposed mechanism via various request sizes.

As the figure shows, the overhead incurred by GPU virtualization is insignificant for Curve25519. For Curve25519, the average degraded throughput is up to 92% of the original throughput. This implies that our mechanism brings negligible performance degradation for asymmetrical cryptographic operations.

However, there is much lower performance for AES-128, which is up to 70% throughput degradation with 3 MB input data. We tested the performance penalty of kernel execution excluding the time of data copying. The degraded throughput is nearly 98% of native throughput. From our point of view, the

time of data copying takes up most of the time of virtual cryptographic computing. This implies that our solution might be ineffective to frequently request for symmetrical cryptographic operations with large input data.

Note that we have not optimize implementation of our prototype yet. The efficiency of data copying via virtio needs to be optimized in our future work.

5 Security Analysis

In this section, our experiment shows our mechanism is able to protect sensitive information from memory disclosure attacks. To this end, we need to make sure there is no occurrence of keys of Curve25519 in memory space of VM. By using **dump-guest-memory** and **info registers** command in QEMU console, the whole memory image and states of registers of VM can be obtained respectively. Since we have already know the plaintext of secret keys in cryptographic computation, we search the secret key strings and the master key string inside the dump file. Fortunately, it turns out that no binary sequence of any key exists. Hence, our GPU virtualization prototype can effectively prevent private keys from leakage of GPU cryptographic computation.

In order to maintain all loaded secrets on GPU and make sure exclusive control of the GPU, PixelVault forces a CUDA kernel to run indefinitely and consumes all available device resources. As a result, PixelVault is dedicated to a single cryptographic operation. Not to say consuming considerable power, this not only degrades performance but also significantly reduces flexibility of computations. Since GPU registers can not be shared between different threads, PixelVault also increases complexity of GPGPU programming.

Compared with the prior works, our solution has the following advantages:

1. We isolate the master key in host. The secret keys used by GPU-accelerated cryptographic operations are not exposed to attackers in plaintext, but encrypted by master key beforehand. However the encrypted secret keys are only decrypted in host OS. Any accelerating cryptographic operation in VM can not reveal the sensitive information.
2. Any authenticated cryptographic computation application can be executed in the VM with a insignificant degradation of throughput. Experiment results show that performance penalty of API-remoting-based GPU computation for ECC is within 8% of native GPU computation.
3. We provide wrappered CUDA runtime library which keeps the same interface as the original library so that developers do not need to modify applications greatly.
4. Our solution does not depend on the characteristics of GPU hardware. Ignoring the architecture of underlying hardware, it is compatible for multiple products.

6 Conclusion

In this paper, we have proposed the design and implementation of a framework for preventing private keys from leakage in accelerating cryptographic computations utilizing GPU virtualization. By isolating the master key in VMM and establishing authenticated GPU binaries, the real keys are never exposed to the guest OS, so that the compromise of the guest OS will not threaten the secrecy of keys. Moreover, The API-remoting-based GPU virtualization with `virtio` is proved as a high performance computing solutions which allows cryptographic applications within VM to leverage GPU acceleration. Our evaluations show that GPU virtualization incurrs a insignificant performance degradation for asymmetric cryptographic algorithm ECC. We also prove that secret keys are never leaked into memory space of VMs. However the major reason of performance degradation incurred by GPU virtualization is data transmission which is a unavoidable problem. Our work currently may not be proper for symmetrical cryptographic operations like AES with large input data.

In the future work, we continue to optimize our prototype with data transmission, lazy calling and implement more runtime APIs. Now our prototype does not support multiplexing and live migration that we intend to extend the prototype with.

References

1. Blass, E.-O., Robertson, W.: Tresor-hunt: attacking CPU-bound encryption. In: Proceedings of the 28th Annual Computer Security Applications Conference, pp. 71–78. ACM (2012)
2. Di, B., Sun, J., Chen, H.: A study of overflow vulnerabilities on GPUs. In: Gao, G.R., Qian, D., Gao, X., Chapman, B., Chen, W. (eds.) NPC 2016. LNCS, vol. 9966, pp. 103–115. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47099-3_9
3. Duato, J., Pena, A.J., Silla, F., Mayo, R., Quintana-Orti, E.S.: Performance of CUDA virtualized remote GPUs in high performance clusters. In: 2011 International Conference on Parallel Processing, pp. 365–374, September 2011
4. Erb, C., Collins, M., Greathouse, J.L.: Dynamic buffer overflow detection for GPGPUs. In: Proceedings of the 2017 International Symposium on Code Generation and Optimization, pp. 61–73. IEEE Press (2017)
5. Garmany, B., Müller, T.: Prime: private RSA infrastructure for memory-less encryption. In: Proceedings of the 29th Annual Computer Security Applications Conference, ACSAC 2013, pp. 149–158. ACM, New York (2013)
6. Guan, L., Lin, J., Luo, B., Jing, J.: Copker: computing with private keys without RAM. In: NDSS, pp. 23–26 (2014)
7. Guan, L., Lin, J. 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), pp. 3–19. IEEE (2015)
8. Halderman, J.A., et al.: Lest we remember: cold-boot attacks on encryption keys. Commun. ACM **52**(5), 91–98 (2009)

9. Harrison, O., Waldron, J.: Efficient acceleration of asymmetric cryptography on graphics hardware. In: Preneel, B. (ed.) AFRICACRYPT 2009. LNCS, vol. 5580, pp. 350–367. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02384-2_22
10. Hong, C.-H., Spence, I., Nikolopoulos, D.S.: GPU virtualization and scheduling methods: a comprehensive survey. ACM Comput. Surv. (CSUR) **50**(3), 35 (2017)
11. Iwai, K., Kurokawa, T., Nisikawa, N.: AES encryption implementation on CUDA GPU and its analysis. In: 2010 First International Conference on Networking and Computing (ICNC), pp. 209–214. IEEE (2010)
12. Jang, K., Han, S., Han, S., Moon, S.B., Park, K.: SSLShader: Cheap SSL acceleration with commodity processors. In: NSDI (2011)
13. Kim, Y., et al.: On-demand bootstrapping mechanism for isolated cryptographic operations on commodity accelerators. Comput. Secur. **62**, 33–48 (2016)
14. Kocher, P., et al.: Spectre attacks: exploiting speculative execution. ArXiv e-prints, January 2018
15. Lee, S., Kim, Y., Kim, J., Kim, J.: Stealing webpages rendered on your browser by exploiting GPU vulnerabilities. In: 2014 IEEE Symposium on Security and Privacy (SP), pp. 19–33. IEEE (2014)
16. Lin, J., Luo, B., Guan, L., Jing, J.: Secure computing using registers and caches: the problem, challenges, and solutions. IEEE Secur. Priv. **14**(6), 63–70 (2016)
17. Lipp, M., et al.: Meltdown. ArXiv e-prints, January 2018
18. Lombardi, F., Pietro, R.D.: Towards a GPU cloud: benefits and security issues. In: Mahmood, Z. (ed.) Continued Rise of the Cloud. CCN, pp. 3–22. Springer, London (2014). https://doi.org/10.1007/978-1-4471-6452-4_1
19. Maurice, C., Neumann, C., Heen, O., Francillon, A.: Confidentiality issues on a GPU in a virtualized environment. In: Christin, N., Safavi-Naini, R. (eds.) FC 2014. LNCS, vol. 8437, pp. 119–135. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-45472-5_9
20. Miele, A.: Buffer overflow vulnerabilities in cuda: a preliminary analysis. J. Comput. Virol. Hacking Tech. **12**(2), 113–120 (2016)
21. Müller, T., Dewald, A., Freiling, F.C.: AESSE: a cold-boot resistant implementation of AES. In: Proceedings of the Third European Workshop on System Security, pp. 42–47. ACM (2010)
22. Müller, T., Freiling, F.C., Dewald, A.: Tresor runs encryption securely outside RAM. In: USENIX Security Symposium, vol. 17 (2011)
23. Pan, W., Zheng, F., Zhao, Y., Zhu, W.-T., Jing, J.: An efficient elliptic curve cryptography signature server with GPU acceleration. IEEE Trans. Inf. Forensics Secur. **12**(1), 111–122 (2017)
24. Di Pietro, R., Lombardi, F., Villani, A.: CUDA leaks: a detailed hack for CUDA and a (partial) fix. ACM Trans. Embedded Comput. Syst. (TECS) **15**(1), 15 (2016)
25. Shi, L., Chen, H., Sun, J., Li, K.: vCUDA: GPU-accelerated high-performance computing in virtual machines. IEEE Trans. Comput. **61**(6), 804–816 (2012)
26. Simmons, P.: Security through amnesia: a software-based solution to the cold boot attack on disk encryption. In: Proceedings of the 27th Annual Computer Security Applications Conference, pp. 73–82. ACM (2011)
27. Stewin, P., Bystrov, I.: Understanding DMA malware. In: Flegel, U., Markatos, E., Robertson, W. (eds.) DIMVA 2012. LNCS, vol. 7591, pp. 21–41. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37300-8_2
28. Szekeres, L., Payer, M., Wei, T., Song, D.: Sok: eternal war in memory. In: 2013 IEEE Symposium on Security and Privacy (SP), pp. 48–62. IEEE (2013)

29. Vasiliadis, G., Athanasopoulos, E., Polychronakis, M., Ioannidis, S.: PixelVault: using GPUs for securing cryptographic operations. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, pp. 1131–1142. ACM (2014)
30. Wartell, R., Mohan, V., Hamlen, K.W., Lin, Z.: Binary stirring: self-randomizing instruction addresses of legacy x86 binary code. In: Proceedings of the 2012 ACM Conference on Computer and Communications security, pp. 157–168. ACM (2012)
31. Zhou, Z., Diao, W., Liu, X., Li, Z., Zhang, K., Liu, R.: Vulnerable GPU memory management: towards recovering raw data from GPU. Proc. Priv. Enhancing Technol. **2017**(2), 57–73 (2017)
32. Zhu, Z., Kim, S., Rozhanski, Y., Hu, Y., Witchel, E., Silberstein, M.: Understanding the security of discrete GPUs. In: Proceedings of the General Purpose GPUs, pp. 1–11. ACM (2017)