LETTER

Rootkit inside GPU Kernel Execution*

Ohmin KWON^{†a)}, Student Member, Hyun KWON[†], and Hyunsoo YOON[†], Nonmembers

SUMMARY We propose a rootkit installation method inside a GPU kernel execution process which works through GPU context manipulation. In GPU-based applications such as deep learning computations and cryptographic operations, the proposed method uses the feature by which the execution flow of the GPU kernel obeys the GPU context information in GPU memory. The proposed method consists of two key ideas. The first is GPU code manipulation, which is able to hijack the execution flow of the original GPU kernel to execute an injected payload without affecting the original GPU computation result. The second is a self-page-table update execution during which the GPU kernel updates its page table to access any location in system memory. After the installation, the malicious payload is executed only in the GPU kernel, and any no evidence remains in system memory. Thus, it cannot be detected by conventional rootkit detection methods.

key words: graphics processing unit, security, rootkit

1. Introduction

A rootkit is installed into a target system, after which it controls the compromised system without being detected. Research on rootkits has taken two paths: i) stealthy features for attacks, and ii) defense strategies against rootkits. In terms of stealthy features, prior works utilized the memory space of the hypervisor [1] and the system management mode [2] to hide the rootkit from the operating system. In addition, other research works proposed the installation of a rootkit in the firmware for hardware devices such as hard drives [3] and network cards [4]. In terms of defense against rootkits, a physical memory scanner can be applied to detect rootkits in system memory. Moreover, to prevent such malicious firmware updates, an update is not allowed when the updated code is not signed with a proper private key. Recent rootkit research, Pikit [5], showed that the vulnerability of the routing mechanism in a NUMA (non-uniform memory access) server can be exploited to install a rootkit. It also introduced several defense solutions.

Our research target for rootkit installation is the dedicated GPU (graphics processing unit) which is connected to the PCIE (peripheral component interconnect express) interface. GPUs were originally designed for graphical data processing, but currently GPGPU platform provides the paral-

lel computing capability of the GPU to various applications, such as deep learning and cryptographic computation. The GPU driver running in kernel mode manages the computing resources of the GPU, and the GPU kernel launched by the CPU process computes the requested work. Through the development of functionalities for GPU kernel executions, the GPU kernel can freely access pre-allocated system memory using the DMA (direct memory access) engine of the GPU. The GPU driver allows the GPU kernel to access the memory space only for the CPU process which launches the GPU kernel. This prevents access to critical memory spaces such as the kernel code and data structure from the GPU kernel.

However, recent studies [6], [7] have shown that a privileged attacker can utilize a dedicated GPU maliciously. Previous work [6] showed that an attacker can gather key log information by launching periodically a malicious GPU kernel which reads the keyboard buffer in system memory. To implement the GPU-based keylogger, a LKM (loadable kernel module) initially manipulates the PT (page table) of the CPU process to create a mapping for the physical memory page of the keyboard buffer. The GPU driver then allocates the memory page of the CPU process to the GPU kernel. Other work [7] showed that malicious GPU firmware** or a driver update can allow the GPU to access any area in system memory.

However, both previous works [6], [7] have the following limitations in the form of evidence left in system memory and possible access only to a restricted area of system memory. They require a keep-alive CPU process which periodically launches a malicious GPU kernel. In addition, the malicious GPU kernel can access only a pre-allocated system memory area even with the support of a LKM load or GPU driver update, and these actions also result in the modification of the kernel area in system memory. To eliminate these limitations for a stealthy rootkit, the proposed method does not leave any evidence and can access any area in system memory.

In this work, we propose a rootkit installation method in the GPU context of a running GPU kernel. GPU context information for GPU kernel execution, such as the GPU code and GPU PT, is created in GPU memory by control of the GPU driver. To install the rootkit in the GPU context, two main methods are introduced here. The installation modifies only the GPU context which resides in GPU mem-

Manuscript received May 23, 2019.

Manuscript publicized August 19, 2019.

[†]The authors are with the School of Computing, Korea Advanced Institute of Science and Technology, Korea.

*This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (NRF-2017R1A2B4006026).

a) E-mail: ohmin59@kaist.ac.kr

DOI: 10.1587/transinf.2019EDL8104

**From Maxwell architecture GPUs, the GPU firmware update is blocked if a verifiable signature is not provided.

ory and thus does not leave any evidence in system memory. We implemented the rootkit installer and payload for privilege escalation of a bash-shell process for a specific user. Through implementation and evaluation, we confirmed that the rootkit inside the GPU kernel for image classification and cryptocurrency mining operates stealthily with negligible performance degradation.

The remainder of the paper is organized as follows. In Sect. 2, we introduce our rootkit installation method. Section 3 presents the implementation of the rootkit installer and payload for privilege escalation attack. In Sect. 4, we describe defense solutions against rootkits inside the GPU kernel. Section 5 concludes the paper.

2. Proposed Method

Our target system continues to utilize GPU computations for high-performance applications such as deep learning and cryptographic operations. In order to utilize GPU computations, the GPU executable code is loaded and the GPU PT structure is created in GPU memory under control of the GPU driver. Subsequently, a CPU process can request a GPU computation by calling a GPU kernel function. The GPU kernel then processes the data already loaded in GPU memory or which resides in the data area of the CPU process.

To install a rootkit inside a GPU kernel execution, an attacker needs a privilege which allows the access to MMIO (memory-mapped I/O) space reserved for the GPU. The privileged installer process, which is launched by the attacker, can access all of the physical memory of the GPU through the MMIO space [8]. As shown in Fig. 1, it searches for the target GPU code and PT and then manipulates these elements in the GPU memory. After finishing this operation, the installer terminates and leaves no evidence in system memory, and only the GPU context information is manipulated in GPU memory. Section 2.1 describes the hijacking of the GPU kernel execution, which is the method by which the stealthy rootkit is installed. Section 2.2 describes the self-PT update execution which can access any location in both the system memory and GPU memory.

2.1 Hijacking of a GPU Kernel Execution

To install a rootkit payload in a GPU execution stealthily, we propose a method of hijacking the execution of the GPU kernel without affecting the original result of the GPU kernel execution. All of the GPU code of the kernel functions to be called is preloaded into the GPU memory. Thus, payload installation is possible by the following steps. First, the installer injects the rootkit payload into empty space in the GPU code area. Next, it overwrites EXIT instructions with JUMP instructions which direct the execution flow to the start point of the injected payload, as shown in Fig. 2. The GPU kernel, which is called after the installation, then runs the injected payload before its execution is terminated. The payload consists of instructions which are executable only in

Fig. 1 Overview of installation method

Fig. 2 Code manipulation for hijacking

the GPU; thus, conventional malware detection mechanisms cannot detect this type of malicious payload.

2.2 Self-PT Update Execution

GPU kernel execution supports virtual-to-physical address translation for memory access through the GPU MMU (memory management unit), including the TLB (translation lookaside buffer) and PT. It accesses physical locations in system memory or GPU memory if valid mapping exists in the GPU MMU. Otherwise, GPU kernel execution terminates with an error report describing the invalid memory access. When a GPU PT is created under the control of the GPU driver, the GPU kernel is allowed to access only the physical memory space including the code and data areas in GPU memory and the data area of the CPU process in system memory.

Our self-PT update execution allows the GPU kernel to access any physical location in both the system memory and GPU memory, as shown in Fig. 3. It grants the ability to access and manipulate critical memory objects such as the kernel code and data to the GPU rootkit. To create the self-PT update execution environment, the installer initially creates new virtual-to-physical memory mapping for the GPU physical memory page in which the PT is located. The installer injects the corresponding PTE (page table entry) into the PT structure in the GPU memory, after which the GPU kernel is able to access its PT. Next, the installer injects the payload of the GPU code which updates the PT.

Fig. 3 Overview of self-PT update execution

Then, GPU kernel execution can create a new mapping for any physical memory page and access the memory page.

After a new mapping for physical memory access is created by the payload, the memory access must direct to the correct physical location. To guarantee correctness of memory access, the virtual address range previously accessed must not be reused owing to TLB caching. Even if the PTE of the accessed virtual address range is updated to remap the address to a new physical memory location, the previous mapping stored in the TLB is not changed, and the access through the reused virtual address directs to the physical memory location of the mapping stored in the TLB, not the updated PT.

3. Implementation

The environment of the target system is Ubuntu 16.04 with the NVIDIA GPU driver 384.130 and version 10 of CUDA (compute unified device architecture). It is equipped with a GTX1070 Pascal GPU. We operated two victim GPU-based systems in two cases. The first is a deep learning inference system that receives image sets and responds by sending classification results. It operates an InceptionV3 model on the Tensorflow platform with the ImageNet ILSVRC2012 dataset. The second is a system that continuously calculates hash values for the mining of a cryptocurrency. For this system, we start the EWBF CUDA Equihash Miner application for BTG (Bitcoin Gold) mining.

3.1 Rootkit Installer

We implemented a rootkit installer that works on both victim systems. The *setuid* of the installer is root privilege to access the GPU MMIO space. It opens */dev/mem* and creates memory mappings for the GPU MMIO. Subsequently, it can access all physical memory of the GPU. As described in Sect. 2, it injects the rootkit payload into the code area and modifies the existing EXIT instructions with a JUMP instruction which jumps to the payload. Next, the installer injects a PTE to create a memory mapping for the physical memory of the PT for the self-PT update execution. After these steps, the installer no longer needs to run and thus terminates its execution.

Table 1 Performance evaluation result. It represents the average of 10 measured values.

Victim system	Without rootkit	Base number for rootkit execution			
		1	10	100	1000
Image inference (img/s)	44.4	25.7 (57.8%)	40.0 (90.0%)	42.9 (96.6%)	43.1 (97.0%)
BTG mining (sol/s)	44.2	42.4 (95.9%)	43.4 (98.1%)	43.5 (98.4%)	43.7 (98.8%)

3.2 Rootkit Payload

The GPU executable code can be implemented in CUDA programming language, and the NVCC (NVIDIA CUDA compiler) helps to generate the GPU binary code. We implemented a 7168-byte GPU payload that scans the memory space and modifies the target kernel data structure to escalate the privilege of the bash-shell process which is launched by a specific user. When the GPU kernel execution is hijacked to the injected payload, the payload initially searches the credential data structure of the process in system memory. Multiple GPU threads find the value of *user_ns*[†] included in the credential data structure. Next, the GPU thread which finds the credential data structure checks whether the value of *user_id* belongs to the target user. It then changes the value to zero, which grants root privilege. Finally, the payload updates the PT to create a mapping for the system memory area to be searched in next execution, after which its execution terminates.

3.3 Performance

We confirmed that the hijacking of the GPU kernel execution and the execution of the rootkit payload do not influence the original GPU computation result for deep learning image inference or cryptocurrency mining. However, the GPU execution performance is degraded. Accordingly, it is possible periodically to check the GPU execution time and determine the existence of a rootkit as time elapses.

To minimize the performance degradation of GPU computation after the installation of the rootkit, the following methods can be applied. The first of these is to reduce the execution time of the payload. To do this, our payload is implemented to access only a contiguous 2 MB area of system memory during a single execution. Next, in order to reduce the number of executions, a malicious action is allowed only if certain conditions are met. In our payload, the number of execution hijacking is counted, and the tasks of searching and manipulating the credential data structure operate only if the count number is a multiplier of a specific base number. This method can limit the number of malicious payload executions by means of a larger base number. Table 1 shows the performance evaluation result with various base numbers for the rootkit payload execution. When

[†]The value of *user_ns* is determined when booting, and it can be found in */proc/kallsyms*.

the execution of a malicious payload is performed only once during 1000 hijackings, the performance outcomes are 97% for image inference and 98% for BTG mining when compared to the original performances without a rootkit.

4. Defense

We classify the possible solutions as hardware-based solutions and software-based solutions and propose a software-based solution for integrity monitoring of the GPU context.

4.1 Hardware-Based Solution

The IOMMU (I/O memory management unit) prevents access to certain regions of system memory from I/O devices. Proper configuration of the IOMMU can prevent the malicious DMA access by a rootkit payload inside the GPU kernel. This requires setup during BIOS execution and support from the kernel. However, the IOMMU is generally disabled in high-performance systems, as it degrades the I/O performance and does not support I/O features such as RDMA (remote DMA) for memory copying between I/O devices. In addition, previous works [7], [9] demonstrated several methods which are able to trick the IOMMU mechanism.

The integrity of the GPU context should be maintained to prevent rootkit installation inside the GPU kernel. Recent research [10] has proposed a new GPU architecture which supports TEE (trusted execution environment) on the GPU. It offloads the GPU context management of the GPU driver to a command processor inside the GPU and configures the PCIe engine of the GPU to restrict MMIO access from privileged CPU processes. Through the above changes in the GPU architecture, context manipulation by a privileged attacker is blocked. However, currently released GPUs do not support the TEE and are thus vulnerable.

4.2 Software-Based Solution

We propose a solution that continually verifies the integrity of the GPU context in GPU memory to detect malicious GPU context manipulations. After the initialization of the GPU context, the monitoring process calculates the hash value of the fresh GPU context, such as the code and PT, and stores these values. It then monitors the integrity of the GPU context by comparing the stored value with the newly obtained value. When the comparison result is not the same, it detects the malicious GPU context manipulation.

The monitoring process is assumed to be protected within a TEE platform such as SGX (software guard extensions), as a privileged attacker can attempt to cause the monitoring process to malfunction before the installation of the rootkit. We implemented a monitoring process which monitors the PT and code used to run the GPU kernel and then evaluated the performance degradation with such monitoring. The performance outcomes were 43.2 (img/s) for im-

age inference and 43.9 (sol/s) for BTG mining. These are respectively 97.2% and 99.3% of the original performance level, indicating that the performance degradation is negligible.

5. Conclusion

This paper presents a new type of rootkit which is installed inside a GPU kernel execution. We described how to install the rootkit by manipulating the GPU code and PT in GPU memory. After the installation, no evidence remains in system memory, and the rootkit payload is executed only on the GPU side. We implemented a rootkit installer and payload which escalates the privilege for a bash-shell process of the target user and demonstrated that it operates stealthily on deep learning inference and cryptocurrency mining systems. We also discussed possible solutions to prevent and detect such instances of rootkit installation. This work emphasizes the importance of GPU security by showing that the GPGPU platform provided for easy use of GPU computations can be a target for rootkit installations.

References

- [1] S.T. King and P.M. Chen, "Subvirt: implementing malware with virtual machines," 2006 IEEE Symposium on Security and Privacy (S&P'06), pp.314–327, May 2006.
- [2] S. Embleton, S. Sparks, and C.C. Zou, "Smm rootkit: A new breed of os independent malware," *Security and Communication Networks*, vol.6, no.12, pp.1590–1605, 2013.
- [3] J. Zaddach, A. Kurmus, D. Balzarotti, E.-O. Blass, A. Francillon, T. Goodspeed, M. Gupta, and I. Kotsidas, "Implementation and implications of a stealth hard-drive backdoor," *Proc. 29th Annual Computer Security Applications Conference, ACSAC '13*, pp.279–288, ACM, 2013.
- [4] S. Sparks, S. Embleton, and C.C. Zou, "A chipset level network backdoor: Bypassing host-based firewall & IDS," *Proc. 4th International Symposium on Information, Computer, and Communications Security, ASIACCS '09*, pp.125–134, ACM, 2009.
- [5] W. Song, H. Choi, J. Kim, E. Kim, Y. Kim, and J. Kim, "Pikit: A new kernel-independent processor-interconnect rootkit," *25th USENIX Security Symposium (USENIX Security 16)*, pp.37–51, 2016.
- [6] E. Ladakis, L. Koromilas, G. Vasiladis, M. Polychronakis, and S. Ioannidis, "You can type, but you can't hide: A stealthy GPU-based keylogger," *Proc. 6th European Workshop on System Security (EuroSec)*, 2013.
- [7] Z. Zhu, S. Kim, Y. Rozhanski, Y. Hu, E. Witchel, and M. Silberstein, "Understanding the security of discrete GPUs," *Proc. General Purpose GPUs, GPGPU-10*, pp.1–11, ACM, 2017.
- [8] C. Maurice, C. Neumann, O. Heen, and A. Francillon, "Confidentiality issues on a GPU in a virtualized environment," *Financial Cryptography and Data Security, Lecture Notes in Computer Science*, vol.8437, pp.119–135, Springer Berlin Heidelberg, 2014.
- [9] P. Stewin and I. Bystrov, "Understanding DMA malware," *Detection of Intrusions and Malware, and Vulnerability Assessment, Lecture Notes in Computer Science*, vol.7591, pp.21–41, Springer Berlin Heidelberg, 2013.
- [10] S. Volos, K. Vaswani, and R. Bruno, "Graviton: Trusted execution environments on GPUs," *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pp.681–696, 2018.