COMPUTERS & SECURITY 62 (2016) 33–48

Available online at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/cose

Computers & Security

On-demand bootstrapping mechanism for isolated cryptographic operations on commodity accelerators

Yonggon Kim [a], *Ohmin Kwon* [a], *Jinsoo Jang* [b], *Seongwook Jin* [a], *Hyeongboo Baek* [a], *Brent Byunghoon Kang* [b,*], *Hyunsoo Yoon* [a]

[a] *Division of Computer Science, School of Computing, Korea Advanced Institute of Science and Technology (KAIST), 291 Daehak-ro, Yuseong-gu, Daejeon, South Korea*

[b] *Graduate School of Information Security, School of Computing, Korea Advanced Institute of Science and Technology (KAIST), 291 Daehak-ro, Yuseong-gu, Daejeon, South Korea*

ARTICLE INFO

Article history:
Received 29 February 2016
Received in revised form 30 May 2016
Accepted 27 June 2016
Available online 1 July 2016

Keywords:
Secure systems
Trusted computing technology
Trustworthy execution
GPU security
GPGPU
Cryptographic key protection
SMM

ABSTRACT

General-Purpose computing on a Graphics Processing Unit (GPGPU) involves leveraging commodity GPUs as massively parallel processing units. GPGPU is an emerging computing paradigm for high-performance and data-intensive computations such as cryptographic operations. Although GPGPU is an attractive solution for accelerating modern cryptographic operations, the security challenges that stem from utilizing commodity GPUs remain an unresolved problem. In this paper, we present an On-demand Bootstrapping Mechanism for Isolated cryptographic operations (OBMI). OBMI transforms commodity GPUs into a securely isolated processing core for various cryptographic operations while maintaining cost-effective computations. By leveraging System Management Mode (SMM), a privileged execution mode provided by x86 architectures, OBMI implements a program and a secret key into the GPU such that they are securely isolated during the acceleration of cryptographic operations, even in the presence of compromised kernels. Our approach does not require an additional hardware-abstraction layer such as a hypervisor or micro-kernel, and it does not entail modifying the GPU driver. An evaluation of the proposed OBMI demonstrated that even adversaries with kernel privileges cannot gain access to the secret key, and it also showed that the proposed mechanism incurs negligible performance degradation for both the CPU and GPU.

1. Introduction

Cryptography has become an essential component in modern computer systems. As the amount of data requiring protection has increased due to the growing importance of security and privacy, the heavy computational workload of cryptographic operations has also become a challenging problem. To alleviate the performance bottleneck affecting modern cryptography, several previous works have suggested harnessing many-core accelerators such as Graphics Processing Units (GPUs), which can be used as massively parallel architectures. Recent GPUs offer significant improvements in throughput and performance-per-watt compared to commodity CPUs (Abe et al., 2012). Leveraging such benefits, several researchers have

* *Corresponding author.*
E-mail address: brentkang@kaist.ac.kr (B.B. Kang).
http://dx.doi.org/10.1016/j.cose.2016.06.006

shown that versatile modern cryptographic operations can be accelerated efficiently using commodity GPU devices (Harrison and Waldron, 2009; Lee et al., 2015; Manavski, 2007; Wang et al., 2014; Zheng et al., 2014). Furthermore, research shows that GPU-accelerated cryptography can be a cost-effective solution to real-world cryptographic implementations such as the Secure Sockets Layer (SSL) and Transport Layer Security (TLS) (Jang et al., 2011).

Unfortunately, GPUs have several security problems. A recent study shows that GPU data stored in GPU memory can be retrieved from different processes, because the GPU does not flush its memory after its termination (Lee et al., 2014; Pietro et al., 2016). Even when implementing an appropriate flushing mechanism for the GPU, malicious users with kernel privileges can easily access the GPU device memory through MMIO (Maurice et al., 2014). Various memory-disclosure attacks (Heartbleed; Blass and Robertson, 2012) and kernel-compromising attacks show that the kernel-enforced security mechanisms can be bypassed. If an attacker can manipulate the kernel or the driver, the secret key can be stolen whenever it is exposed in the GPU device memory.

To protect the secret key under a broad range of GPU vulnerabilities and threats to the underlying host system, we propose a low-cost key-protection mechanism for general-purpose computing on a GPU (GPGPU). We leverage System Management Mode (SMM) and an SMM-based handler function to implement secure key management, along with a bootstrapping mechanism that enables the key-protected execution of GPU-accelerated cryptography. SMM is a privileged execution mode for x86 architectures offered by commodity CPUs. In SMM, only the authorized handler – the so-called System Management Interrupt (SMI) handler – can be executed. Other processes, including malicious processes, cannot be executed in SMM. Because the OS kernel and any malicious processes are unaware of the SMM execution, we can implement an SMI handler that transparently manages the GPU device.

By implementing a simple bootstrapping mechanism within the SMI handler, we securely upload the secret key into the GPU cache. Unlike the GPU device memory, even privileged CPU processes cannot access the GPU cache. There are multiple types of GPU caches, and the GPU constant cache can be easily utilized as a key storage by exclusive use of the GPU constant memory for the secret key. When the SMI handler clears the remaining footprint of the secret key in the device memory, the secret key is isolated within the GPU constant cache. Similarly, we can also isolate the entire GPU code at the GPU instruction cache to prevent any control-flow modifications of the GPU program.

The main challenge to implementing the bootstrapping mechanism arises because, although the SMI handler can securely access the GPU device, it is unable to utilize the existing GPU driver and OS kernel. This is because the OS kernel and GPU driver are suspended in SMM. Modern GPUs are complex, and the GPU driver is responsible for controlling the underlying hardware. Without the help of the GPU driver, engineering efforts to implement the bootstrapping process increase dramatically. To overcome this challenge, we split all the GPU control logics needed for bootstrapping mechanism into two classes of tasks: security-sensitive tasks, and security-insensitive tasks. Consequently, security-insensitive tasks can be handled by the existing GPU driver and rudimentary OS functionality. We devised two consecutive steps for the bootstrapping process: one step in normal CPU mode (i.e., protected mode), and the other in SMM. By delegating most of GPU control logic (i.e., security-insensitive tasks) to the first step, we can significantly reduce the complexity of the control logic required for the SMI handler.

With our bootstrapping mechanism, the secret keys used by GPU-accelerated cryptography are not exposed to attackers. Before bootstrapping, the secret keys are stored in the protected memory space, which is only accessible by the SMI handler. In SMM, the secret key and GPU program are safely uploaded to the GPU caches. Although processor environments isolated in SMM are only momentarily utilized until the upload is complete, we can preserve the confidentiality of the uploaded key and program during the acceleration of cryptographic operations, since GPU caches are inaccessible from any host processes. If the uploaded GPU program terminates, all content in the GPU caches is invalidated. Thus, any subsequent GPU program cannot retrieve the secret keys.

To minimize the performance degradation incurred by the proposed mechanism, we devise several optimizations for efficient bootstrapping. Furthermore, to demonstrate the feasibility of our bootstrapping mechanism, we implemented a prototype using a commodity CPU and GPU. Our prototype includes GPU-accelerated RSA and AES cryptographic operations, and results in minimal performance loss. We carefully evaluated the proposed mechanism in terms of its security, in order to confirm that it is robust even to attackers with kernel privileges.

Our bootstrapping mechanism is advantageous in many ways: (i) transparency – we leverage existing GPU drivers and operating systems, without the need to modify them; (ii) a small TCB – our mechanism adds only a few hundred lines of code for the SMI handler, significantly minimizing the size of the TCB; (iii) compatibility – our mechanism is based on commodity hardware; and (iv) simplicity and speed – the additional code required for bootstrapping is relatively simple compared to other approaches (Sani et al., 2014; Yu et al., 2015).

In particular, our work makes the following contributions:

1. To our best knowledge, we are the first to suggest using the GPU cache to store cryptographic keys, such that the commodity GPU can safely accelerate cryptographic operations without revealing sensitive information.
2. We suggest an SMM-based bootstrapping mechanism, referred to hereafter as the On-demand Bootstrapping Mechanism for Isolated cryptographic operations (OBMI). OBMI securely uploads the secret key into the GPU cache in an on-demand fashion.
3. OBMI includes mechanisms for checking the integrity of the accelerated GPU code. We propose a code-verification mechanism to guarantee that only reliable code can utilize the GPU device for cryptographic key-related operations.
4. We implemented a prototype using a commodity Nvidia GPU. Our evaluation shows that OBMI incurs minimal performance overhead and is scalable to multiple secret keys.
5. By exploring several possible attacks, we show the robustness of OBMI. We demonstrate that our approach enables secure operations on commodity GPUs without increasing the TCB of the system or degrading its overall performance.

The rest of the paper is organized as follows. Section 2 discusses previous related works. Section 3 describes the background for OBMI. Section 4 discusses a threat model and the assumptions of this work. Section 5 gives an overview of OBMI. Section 6 explains how the OBMI uploads the secret key into the GPU cache. Section 7 shows how the OBMI guarantees the integrity of the GPU program. Section 8 shows the evaluation results of OBMI. Section 9 discusses several implementation issues, and Section 10 concludes the paper.

2. Related works

Several previous works exist for leveraging SMM. For example, HyperCheck (Zhang et al., 2014) and HyperSentry (Azab et al., 2010) are SMM-based systems that check the integrity of a hypervisor. Malt (Zhang et al., 2015) is a debugging framework that employs SMM to study malware transparently. Several introspection frameworks are also available that use SMM to inspect the state of a system (Zhang et al., 2013) or device firmware (Zhang et al., 2014). To the best of our knowledge, however, no previous works have examined GPU security and the proper utilization of SMM to realize it.

Besides SMM, several other hardware-assisted trusted execution environments (TEE) are available. Intel developed Trusted eXecution Technology (TXT) (Intel Corporation, 2013), providing a secure way to establish system software such as an OS or hypervisor through the Dynamic Root of Trust for Measurement (DRTM). AMD introduced similar technology, called the Secure Virtual Machine (SVM) (Advanced Micro Devices, 2005). However, the performance loss incurred by TXT and SVM are several orders of magnitude higher than the performance loss from SMM (McCune et al., 2008). Recently, Intel also introduced Software Guard Extensions (SGX) (Intel Corporation, 2014), a safe enclave for executing software. Although SGX ensures a protected execution that is safe from malicious OS kernels, it cannot be utilized to securely manage a GPU, because SGX does not provide access control for I/O devices. TrustZone (ARM, 2009) is a popular TEE widely deployed on mobile devices. However, its underlying architecture is ARM, and almost all dedicated GPUs are based on x86 architectures.

Several researchers inspected attacks and defense mechanisms relating to cryptographic keys. The Heartbleed Bug (Heartbleed), an OpenSSL vulnerability, facilitated memory-disclosure attacks and exposed cryptographic keys stored in system memory. Cold boot attacks (Halderman et al., 2009) allow attackers to retrieve sensitive information within the RAM chips by freezing them. To prevent cold boot attacks, AESSE (Müller et al., 2010), TRESOR (Müller et al., 2011), and Amnesia (Simmons, 2011) store AES keys exclusively in CPU registers. PRIME (Garmany and Muller, 2013) and Copker (Guan et al., 2014) are cold-boot-resistant implementations of the RSA algorithm that leverage the CPU registers and caches, respectively. Recently, Mimosa (Guan, 2015) utilized hardware transactional memory to protect cryptographic keys from cold boot attacks and memory disclosure attacks. Our proposal is significantly different from these solutions in two important ways. First, unlike the above solutions, we do not assume the integrity of the OS kernel. Because a compromised kernel can easily access the register or the cache within a CPU, all of the above solutions are infeasible given a compromised OS kernel. Second, these solutions focus exclusively on CPU computations, and they cannot be extended to GPU-based cryptographic operations.

Recent research has leveraged hypervisor-based solutions to isolate the program execution on untrusted operating systems (Hofmann et al., 2013; McCune et al., 2010). Similarly, hypervisor-based GPU virtualization can be used to isolate the GPU (Suzuki et al., 2014; Tian et al., 2014) and GPU-accelerated cryptographic operations. However, hypervisors require a copious amount of code and are themselves vulnerable to attack (CVE Details). Moreover, a recent study insists that the performance degradation incurred by GPU virtualization is considerable (Liu et al., 2015). Recent works (Sani et al., 2014; Yu et al., 2015) have implemented a security kernel that isolates the GPU from malicious access, offering protected GPU functionalities such as a secure display (Yu et al., 2015) or graphics computation (Sani et al., 2014). However, since this approach entails rewriting the GPU driver within the security kernel, it is inapplicable to Nvidia GPU devices that have not revealed its GPU driver's source code or CUDA platform, making it infeasible to modify and re-implement.

PixelVault (Vasiliadis et al., 2014) proposed a key-protection mechanism utilizing GPU registers. Similar to our proposed solution, PixelVault enables secure cryptographic operations even with a vulnerable OS kernel. In order to utilize PixelVault, however, the GPU must be dedicated to single cryptographic operations. This significantly reduces the flexibility of GPU computations and consumes considerable power. In addition, using GPU registers to store cryptographic keys is impractical with large keys due to its limited size. Moreover, GPU registers can only be manipulated with scalar-type variables and cannot be shared between different threads. This significantly increases the complexity of GPGPU programming. However, unlike PixelVault, we have utilized SMM and the GPU constant cache to protect cryptographic keys while preserving efficiency and programmability.

Other research has investigated GPU vulnerabilities (Lee et al., 2014; Maurice et al., 2014; Miele, 2015; Pietro et al., 2016). We analyzed every threat reported in this research in order to confirm that the proposed mechanism efficiently protects GPU acceleration even in presence of privileged malware.

3. Background

3.1. SMM

SMM (Advanced Micro Devices, 2013; Intel Corporation, 2016) is a special operating mode for x86 CPUs originally designed for system-management tasks such as power management and system hardware control. For isolated execution environment, SMM supports two kinds of protection: temporal and spatial protections. In terms of spatial protection, SMM supports the protected memory area, called SMRAM, to which only SMM has access. Normal tasks, including those of the hypervisor and kernel, cannot access SMRAM. The latest commodity processors support up to 4 GB of SMRAM, which is sufficient for security-related purposes such as OBMI.

In terms of temporal protection, SMM supports exclusive execution. When a special interrupt is called, all cores enter

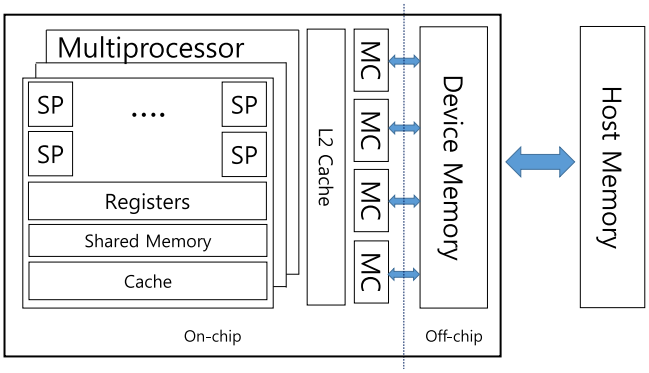

Fig. 1 – GPU architecture and memory hierarchy.

SMM and the SMI handler processes. However, all tasks running in the cores are suspended at the same time. The state of each task is automatically saved within the SMRAM. After the SMI handler finishes, the saved states are restored and suspended tasks are resumed. In addition, all of the caches are flushed when entering and exiting SMM. There are no shared resource between tasks in SMM and normal tasks. With exclusive execution, the SMI handler cannot be interrupted from other tasks, including the kernel and hypervisor.

3.2. *GPU architecture and memory hierarchy*

The overall architecture for a general Nvidia GPU is shown in Fig. 1. We assumed a dedicated GPU on the grounds that these are more common and suitable for high performance computations. The GPU consists of hundreds of processing cores compartmentalized by several streaming multiprocessors (SMs). Each SM contains multiple execution units (known as CUDA cores or stream processors), scheduler units, registers, and caches. GPU devices contain dedicated DRAM memory, and they are controlled through multiple on-chip memory controllers. Massive amounts of data can be transferred between the device memory and the host memory using DMA or MMIO (Fujii et al., 2013).

To enable general-purpose processing on a GPU, Nvidia offers a programming environment called CUDA. In this paper, we utilize the CUDA framework for cryptographic operations on GPUs. Further, we used an Nvidia GPU, a dedicated GPU, for the prototype implementation.

During the execution of the CUDA program, the GPU device concurrently executes the same GPU code (called the GPU kernel), for a vast number of GPU threads. The multi-granularity of the GPU software abstractions is shown in Fig. 2, along with the respective GPU hardware units. The GPU block is a unit for scheduling thread chunks. The GPU blocks are distributed among SMs, and each SM executes only a single GPU block at a time. Within an SM, a chunk of GPU threads is divided into several warps. A warp is the basic unit of execution in an SM, consisting of 32 threads and executed with the same instruction in lockstep. By scheduling multiple warps within a single SM, GPGPU can hide memory latency, thus maximizing the utilization of the processing cores.

Nvidia CUDA offers multiple memory regions with unique characteristics for distinct purposes. There is both on- and off-chip memory for GPUs, as shown in Fig. 1. On-chip memory includes the register and shared memory, whereas off-chip memory includes local, global, constant, and texture memory. Recent research shows that the device memory of GPUs is subject to severe security problems that risk information leakage (Lee et al., 2014; Pietro et al., 2016). This is because the device

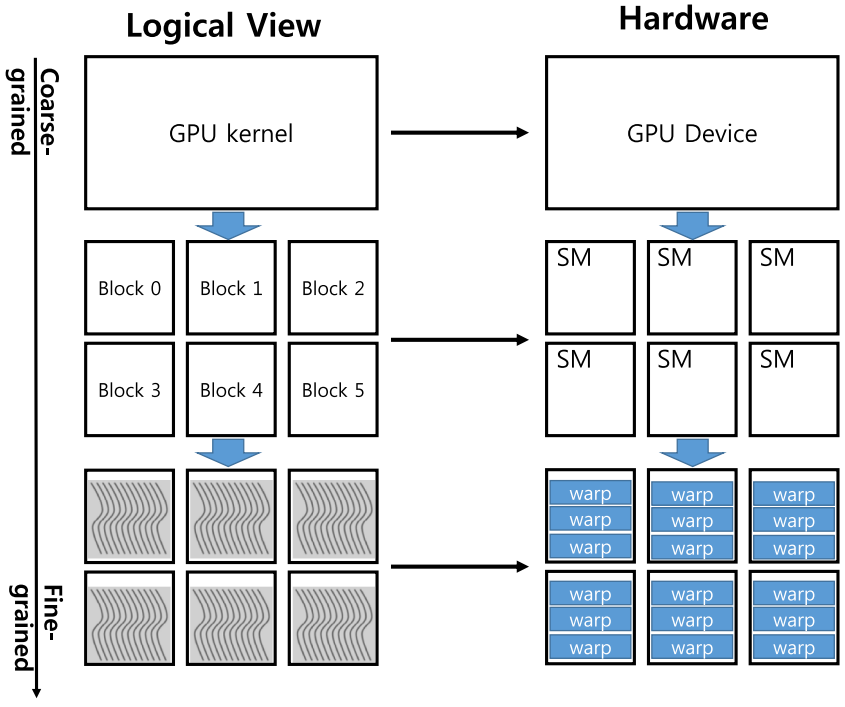

Fig. 2 – GPU abstraction.

memory can be easily accessed by a host process through MMIO. Moreover, GPUs are vulnerable to timing attacks that exploit the inappropriate flush timing of the data in each memory region. The lifespan of the variables stored in GPU memory regions (e.g., shared memory, off-chip memory) might exceed the lifespan of the GPU kernel. In this way, the subsequent GPU kernel can retrieve sensitive information from the previous GPU kernel (Lee et al., 2014; Pietro et al., 2016).

With the Fermi architecture (Wittenbrink et al., 2011), the GPU device adds a data cache consisting of dedicated per-SM cache and per-GPU shared cache, generally referred to as the L1 and L2 cache, respectively. The L1 cache is located within each SM, and used exclusively by each SM. The L2 data cache is utilized by all SMs, and is much larger than the L1 cache. Texture memory and constant memory also have their own cache (Nvidia Corporation, 2015; Wong et al., 2010).

Unlike off-chip memory or shared memory, data in caches have the same lifespan as a GPU kernel. When the GPU kernel terminates, the cache invalidates its content such that next GPU kernel can fill it again with off-chip data through cache misses. Unlike GPU device memory, there is no explicit way to access or manipulate the GPU cache. As a result, the host process or subsequently launched malicious GPU kernel cannot access the data content that is stored and used within the GPU cache. Thus, we leverage the cache to provide safe key storage and cryptographic operations on a GPU. Although a recent proposal (Lee et al., 2014) warns of attacks that can retrieve the content within the data cache for global memory, we utilize only constant cache, which is not vulnerable to such attacks.

3.3. *GPU device control*

The kernel and GPU drivers have the important role of managing and controlling the GPU device, and use the Peripheral Component Interconnect Express (PCIe) standard to interface with it. For Nvidia GPUs, the details for controlling the GPU device with the GPU driver are not documented or exposed, but the GPU driver uses an I/O port or MMIO to access and control the GPU device in the same manner as an ordinary I/O device does. In the reverse-engineering community, considerable effort has revealed the internal mechanism of Nvidia GPUs (Envytools; Nouveau). Based on this, an open-source GPU driver for Nvidia, named nouveau, has been released as an official Linux kernel module. Nouveau is limited in terms of performance (Larabel, 2014) and functionality (e.g., nouveau does not provide support for CUDA), but the efforts devoted to reverse engineering the proprietary driver have uncovered pertinent information regarding the underlying hardware mechanism. The reverse-engineering community enlisted several low-level channels for accessing the GPU device memory, and we leveraged one of these channels for our bootstrapping mechanism.

4. Threat model and assumption

An attacker can manipulate the host memory using various memory-disclosure attacks by exploiting a vulnerability in the software (Heartbleed) or through direct memory access (DMA)

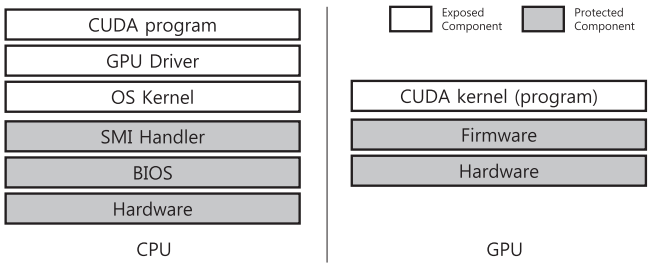

Fig. 3 – Trusted component diagram.

(Blass and Robertson, 2012). We assume that an attacker can compromise the operating system and execute arbitrary privileged code. In this situation, the attacker can modify any CUDA program or cryptographic key, provided that these are located within the host memory. Moreover, malicious users can alter the CUDA API or GPU driver, such that the integrity and confidentiality of GPU executions operated by the GPU driver cannot be guaranteed.

As described in the Section 3 and in previous works (Lee et al., 2014; Maurice et al., 2014; Pietro et al., 2016), the GPU memory region is vulnerable to malicious MMIO access or timing attacks. Content in the GPU register can also be exposed to the device memory through a register-spilling mechanism (Micikevicius, 2011). Since an attacker with kernel privileges can easily exploit these vulnerabilities, we aim to provide a defense mechanism resilient to such vulnerabilities.

By utilizing SMM, we implement a small trusted code, namely the SMI handler, to manage cryptographic keys and operate our bootstrapping mechanism. We exclude the kernel and GPU driver from the TCB. This way, the TCB includes only the BIOS and the underlying hardware, as shown in Fig. 3. During the boot sequence, the SMI handler is installed through the BIOS, and the integrity of the BIOS can be verified with TPM-based attestation. We assume the existence of external administration station, which is responsible for managing important credentials. We also assume that a trusted system administrator will obtain the master key from the external administration station during the secure booting sequence. After this secure boot, the master key is securely stored within the SMRAM, and it can be leveraged to create a secure channel between the SMI handler and the external administration station.

In this paper, we do not deal with data confidentiality or data integrity. We assume that the data are stored and processed within the device memory and that it can be exposed to the attacker. Although it is also possible to secure the data using an additional cryptographic key, we have focused on a mechanism for protecting the cryptographic key and GPU kernel. Moreover, we ignore denial-of-service attacks. Our mechanism does not guarantee the availability of the GPU device or cryptographic operations.

We do not consider physical access to the machine by a malicious user. Attackers with physical access might be capable of subverting the SMRAM with a cold boot attack. This issue is orthogonal to our bootstrapping mechanism, and such an attack on the SMM is beyond the scope of this paper. Subverting the firmware of the device (Zaddach et al., 2013) is another

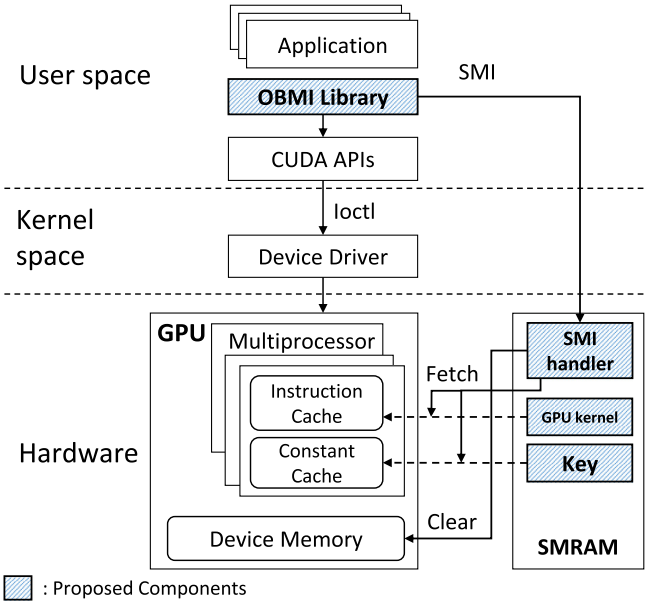

Fig. 4 – Diagram of the overall design of the proposed solution.

attack vector. However, we exclude such attacks in this paper, because the capability of GPU firmware has not yet been sufficiently explored. Furthermore, the recent GPU model offers a security mechanism to protect the GPU firmware from malicious behavior (Falcon Security).

5. Design overview

In general, OBMI controls the GPU in two different ways, as shown in Fig. 4. By using the CUDA APIs, OBMI can utilize the GPU with various API functions in the same manner as a typical GPGPU program. However, security-related tasks cannot be undertaken in this manner, because attackers can compromise the CUDA APIs or device driver with a vulnerable OS kernel. To control the GPU device securely, we suggest an additional channel operated in SMM and triggered by the SMI. Any user application can trigger the SMI through OBMI's library. In this way, users can launch secure cryptographic operations on commodity GPU devices. The main proposed component of our solution is the SMI handler, which safely manages the secret key and initiates the necessary GPU kernel in SMM.

5.1. *Using the GPU device driver*

The first step of our bootstrapping mechanism for securing GPU devices involves the proprietary GPU driver, which has a versatile API for GPGPU. In the case of the Nvidia GPU, for example, various CUDA APIs exist, allowing users to easily manipulate GPU resources. These CUDA APIs provide an easy and efficient way to control the underlying GPU for our bootstrapping mechanism. OBMI delegates several security-unrelated tasks to the CUDA APIs and the underlying proprietary GPU driver. Before accelerating cryptographic operations on the GPU, several ordinary tasks must be completed. These tasks are not directly related to cryptography, but they are essential to GPGPU acceleration. First, the required GPU control structures must be allocated, along with the necessary device memory. Moreover, data must be copied to the device memory, and the GPU kernel must be assigned to the GPU device in order to launch it using a command queue. We leverage the proprietary GPU driver without any modification, and efficiently process all of the above tasks.

5.2. *User library interface*

Any user application requiring the secure acceleration of cryptographic operations can initiate our bootstrapping mechanism through the OBMI library interface. When the proprietary GPU driver properly initializes all GPU resources, the library triggers the SMI, which changes the CPU execution mode to SMM, such that the security-related tasks for our mechanism can be accomplished. In SMM, OBMI uploads the necessary secret key for cryptographic operations into the GPU device, which are initially stored in SMRAM, as shown in Fig. 4. Although the OBMI library can be altered maliciously, all sensitive data are manipulated only in SMM. Therefore, no security vulnerabilities exist.

5.3. *Key-protected execution*

The proposed SMM-based bootstrapping mechanism can securely upload the secret key into the GPU cache that the host cannot access. However, it is useless if the GPU kernel has been compromised when it utilizes the uploaded secret key. Moreover, instructions for the GPU kernel are located in the GPU device memory, to which the host has access. Thus, a malicious GPU code-injection attack is possible during GPU acceleration. In order to guarantee the integrity of the GPU kernel and prevent such an attack, we also leverage SMM to safely isolate the authenticated GPU kernel within the GPU instruction cache. As a result, OBMI allows only two kinds of authenticated programs access to the secret key: the SMI handler, and an authenticated GPU kernel. We discuss this mechanism in detail in Section 7.

The GPU kernel and secret key uploaded by the SMI handler are stored within the instruction cache and the constant cache, respectively. However, because the GPU cache is a hardware-managed structure, there is no way to control internal cache operations such as evictions of cache lines. Fortunately, we observe that the GPU cache is organized such that each different memory type uses its own dedicated cache structure. Instructions for the GPU are thus cached in a dedicated instruction cache that is sufficiently large to store the GPU program and implement various cryptographic operations (Vasiliadis et al., 2014). Moreover, constant memory in the GPU has its own dedicated cache (Wong et al., 2010). We show that the constant cache and instruction cache can store all codes and secret keys for cryptographic operations such as RSA or AES.

After the GPU kernel code and secret key are uploaded, the GPU begins its acceleration, and the CPU exits from SMM. During the GPU acceleration of these cryptographic operations, the CPU can execute any arbitrary malicious code and even access the

GPU device memory. However, attackers doing so cannot obtain any sensitive information, because the SMI handler clears every footprint of the GPU kernel and secret key in the device memory before returning from SMM. The security analysis in Section 8.3 shows that the proposed OBMI mechanism successfully prevents the key from being exposed through several possible attacks. After the cryptographic operation ends and the GPU kernel terminates, the user applications can copy the resulting data from the device memory using the CUDA API. However, the cache-uploaded secret key and GPU kernel are automatically invalidated, such that the attackers are unable to access them.

5.4. *Pre-bootstrapping key management*

The main focus of our paper is the key-protection mechanism during the GPU acceleration of cryptographic operations. However, the key must be properly managed before the acceleration for complete key protection. For this purpose, we leverage the SMRAM and SMI handler in order to perform key management operations safely.

First, we provide a dynamic key-generation API through the OBMI library to generate new cryptographic keys. For this purpose, we implement a pseudo-random function based on SHA256 within the SMI handler. When the SMI handler is called, it returns the key index and stores the newly generated key within the SMRAM. Then, user applications can request any cryptographic operations by pointing out the index of the key. These are only used for limited session time, after which they are securely destroyed by the SMI handler.

In modern cryptography, a cryptographic protocol utilizes a temporary session key. In a protocol such as SSL/TLS, several session keys are newly generated during the handshaking protocol. In the evaluation (Section 8), we show that the SMI handler can also be utilized to provide an efficient and secure key-generation process.

Unlike temporarily used session keys, some secret keys, such as RSA keys, might require long-term secure storage (e.g., across a power cycle). In order to support this kind of secret keys, we assume an external administration station in which any credentials requiring long-term protection are stored. We can also establish a secure channel between the external station and the SMI handler using a master key obtained during secure booting, as described in Section 4. Using the secure channel, credentials initially stored in the external station can be deployed into the SMRAM, such that OBMI can utilize them for accelerated cryptographic operations.

6. Bootstrapping the secret key

In this section, we describe how the GPU cache can be used as a safe place for the secret key. Then, we explain how the secret key is uploaded to the GPU cache in an on-demand fashion.

6.1. *Constant cache: safe key storage within GPUs*

To perform cryptographic operations, the cryptographic key should be stored where the GPU program can access it. Global memory and shared memory are the most common places to store variables, but they are inappropriate for storing private keys, owing to the security flaws discussed in Section 3.

Registers can be used to store private information, and PixelVault has already shown that GPU registers can be used for storing cryptographic keys (Vasiliadis et al., 2014). However, when registers are used to store keys, every single thread should preserve a sufficient number of registers to store the keys, because registers are allocated to each thread independently and cannot be shared among different threads. This risks incurring a significant degradation in performance, because registers have a limited capacity for each multiprocessor. Moreover, it is impossible to leverage a register for key storage when the key size is large. For example, the total key size with our implementation for RSA decryption is 1024 bytes. This exceeds the register limitation per thread – i.e., 63 32-bit registers per thread – on CUDA Compute Capability 2.0.

The GPU cache is a protected space in which (i) no programs running on the CPU can access the cache; and (ii) all content in the cache is inaccessible when the GPU kernel finishes. Thus, we can use the GPU cache as a private key container, provided that we can safely establish the key in the cache and delete any remaining footprints in the device memory. One problem with using the cache as a key container is that the internal hardware mechanism controls the cache-replacement policy, making it difficult to ensure that the key will remain in the cache indefinitely without eviction. Fortunately, constant memory (i.e., memory space for constant variables declared with the type qualifier "__constant__") utilizes a dedicated cache that is separate from other data caches (Cheng et al., 2014; Nvidia Corporation, 2015). Consequently, when programmers use the constant memory exclusively for the secret key, the GPU constant cache can be used as a key container that avoids the problem of cache eviction.

6.2. *On-demand key-uploading process*

One of the requirements for the secure key-uploading process is the guarantee that no malicious process, including the OS kernel, can interrupt the process. To achieve this requirement, we use the SMI handler to guarantee that the entire process takes place in SMM. However, even though the SMI handler has the highest privilege among CPU programs, writing data to the GPU cache is unavailable. Therefore, to write the secret key onto the GPU cache, we use both the SMI handler and an authenticated GPU kernel. The overall process is shown in Fig. 5. First, the SMI handler reads the secret key from the SMRAM. The SMI handler then copies it to the GPU constant memory using MMIO. To fill the GPU constant cache with the secret key, the GPU device executes an authenticated GPU device code that performs read operations on the secret key. Finally, the SMI handler removes the remaining key footprint in the GPU device memory. Because the constant memory is unsafe, all steps are executed in SMM, with the exception of Step 1.

The stub GPU code depicted in Fig. 5 includes nothing other than a loop that waits for the copy of authenticated GPU code. The detailed mechanisms related to injecting and executing authenticated GPU code are described in the next section. Briefly, the GPU driver launches the stub GPU code in normal CPU mode. This stub code exists merely to initialize the CUDA

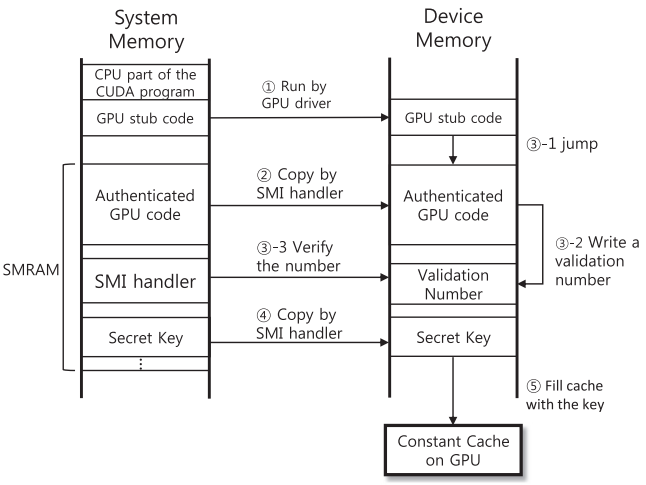

Fig. 5 – Bootstrapping process. Step (1): the GPU driver initially executes a simple GPU stub code. Step (2): the SMI handler injects an authenticated GPU code into the GPU device memory. Step (3): the SMI handler verifies that the injected GPU code is running. Step (4): the SMI handler copies the secret key from SMRAM to the GPU constant memory. Step (5): the GPU executes the injected GPU code such that it fills the GPU constant cache with the secret key. Step (6): the SMI handler removes the footprint of the secret key in the GPU constant memory.

kernel. By launching the stub GPU code with the CUDA API and leveraging the GPU code-injection mechanism described in Section 7.1, we can easily execute arbitrary GPU code even in SMM without any re-implementation of the GPU driver.

After validating the authenticated GPU code (Steps 1 to 3, described in the Section 7), the SMI hander uses MMIO to perform the first copy operation (Step 4). To write the secret key to the constant memory, the exact memory address is needed. Fortunately, we can locate the physical address for the device memory, because it is fixed for constant data. We implemented RSA and AES cryptographic algorithms to determine whether the cryptographic keys are always stored in same physical address in device memory. The results showed that the cryptographic keys are spread throughout the device memory, as shown in Fig. 6, and their locations are always the same. The static behavior of constant memory derives from the non-preemptive execution model of modern GPUs. This means that only a single program utilizes the entire GPU device memory. Moreover, unlike the sophisticated memory management of the OS with CPU memory, it is presumed that GPU device manages device memory with simple memory virtualization, optimized for accelerating a single program. The deterministic property of the constant memory allows us to update its content with simple MMIO operations. We leveraged the PRAMIN region within the GPU MMIO region, which maps a 1 MB segment of the device memory (Maurice et al., 2014; Nouveau). Each device memory region, including the constant memory, can be acquired through PRAMIN by setting its base address through MMIO.

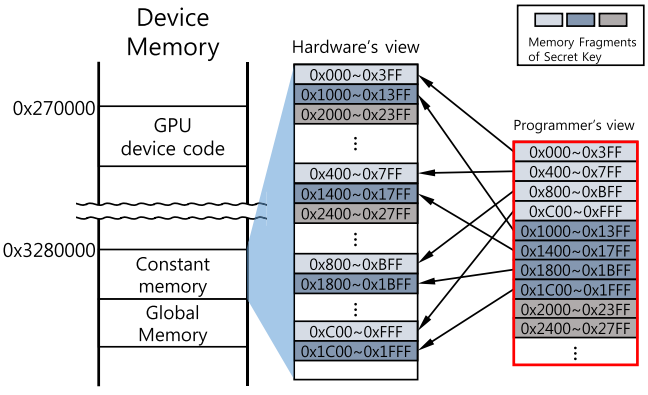

Fig. 6 – Memory layout for the secret key.

Next, the secret key is uploaded into the constant cache by executing the authenticated GPU code (Step 5). The authenticated GPU code is implemented in a way to access each piece of the secret key, thus filling the constant cache with the secret key through the cold cache miss. In our prototype, this authenticated GPU code executes cryptographic operations such as RSA or AES. In doing so, the authenticated GPU code uploads both the secret key and the GPU code itself into the GPU caches. After this uploading process, the authenticated GPU code within the cache benignly executes cryptographic operations leveraging the secret key isolated in the constant cache.

The size of the GPU constant memory is 64 KB, and the size of the GPU constant cache is smaller. The GPU device has multiple levels of the constant cache, and their sizes are 2 KB, 8 KB and 32 KB for the L1, L2, and L3 caches, respectively (Wong et al., 2010). One thing to consider is that the L2 and L3 caches are shared with instruction memory (Wong et al., 2010). Thus, we need to test whether the secret key is evicted during the execution of GPU code when the total size of the secret key exceeds 2 KB, the size of L1 constant cache. Cryptographic operations with a single key do not present a problem, because both RSA and AES use a key size of 1024 bytes and 176 bytes, respectively. However, when cryptographic operations with multiple keys are needed, the limited size of the L1 constant cache is potentially problematic. For AES, this can be especially disadvantageous, since AES operations with multiple keys are common. To evaluate the scalability of our mechanism, we experimented with various numbers of AES keys. Our evaluation shows that the key is evicted only when the total size of the key exceeds 47,520 bytes. This result indicates that we can safely accelerate AES with multiple keys up to 270. Therefore, the proposed mechanism is scalable to multiple keys.

7. Bootstrapping program: running authenticated GPU code

In this section, we describe the bootstrapping process for the authenticated GPU kernel. Executing authenticated GPU code is a prerequisite for bootstrapping the secret key, as described in the previous section. Moreover, the integrity of the GPU kernel must be ensured in order to protect cryptographic keys during accelerated cryptographic operations. Using the proposed mechanism, we can securely execute arbitrary, authenticated GPU code without any tampering. This is

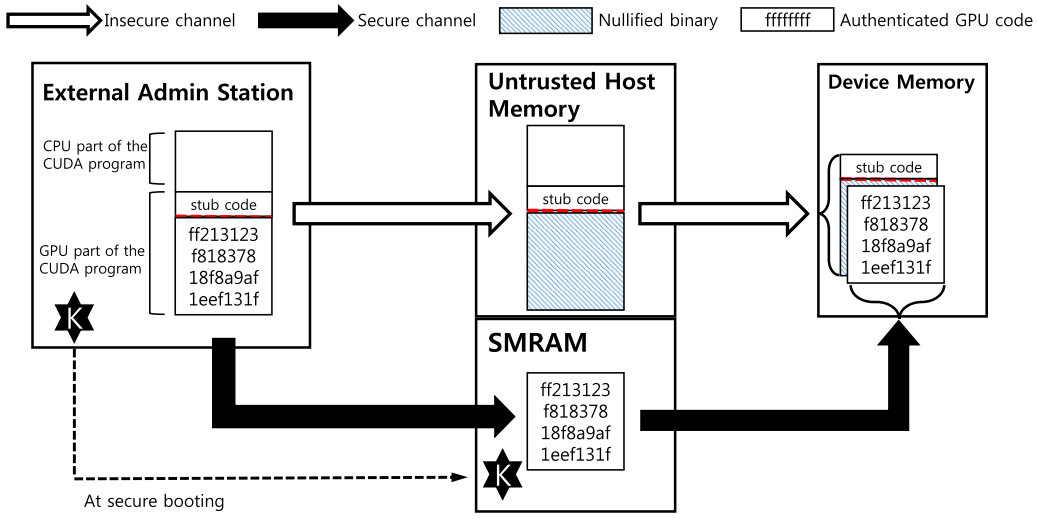

Fig. 7 – Flow diagram for the on-demand injection mechanism for authenticated GPU code. The attacker can only see the nullified GPU binary in the host or device memory.

possible because we upload the entire code into the GPU instruction cache in SMM.

The main challenge to bootstrapping the GPU kernel is to securely upload the entire code into the instruction cache. Whereas the SMI handler can securely upload the GPU kernel, it lacks the functionality to initiate any GPU kernel, because OS and GPU drivers are suspended in SMM. To resolve this problem, we leverage the ordinary GPU driver in normal protected mode. Before triggering the SMI handler, we first execute the GPU program, which contains only the while-loop as a stub GPU kernel. Then, a GPU code-injection mechanism safely injects the authenticated GPU kernel code into the running stub GPU kernel in SMM. Indeed, GPU memory allocation is deterministic, insofar as the same program always allocates the code to the same location. This makes it possible for the SMI handler to append the additional code – e.g., the cryptographic operation – by accessing the GPU device memory with MMIO operations.

In the remainder of this section, we detail the mechanisms for executing the authenticated GPU kernel. First, we describe the on-demand code-injection mechanism for the authenticated GPU kernel. Then, we discuss how it can be determined that the GPU device correctly executes the authenticated GPU kernel, rather than a malicious GPU kernel.

7.1. *Injecting the authenticated GPU kernel*

To simplify the code injection, we create the authenticated GPU code in advance rather than generating it dynamically. First, a complete GPU program is implemented. This program includes both the CPU implementation for resource-management tasks and the GPU kernel for cryptographic operations. Then, we can extract from the GPU kernel binary all instructions related to cryptographic operations. Consequently, we can obtain the collection of code fragments that are later injected. When extracting these code fragments, their exact locations within the GPU instruction memory must be noted, such that we can inject them into the correct location later. These locations are discovered simply by running the GPU kernel, because their locations are always same whenever the same GPU kernel is executed.

Both the implementation of the GPU program and the extraction of the code fragments should be performed in a safe place, such as an external administration station. After the extraction, we simply nullify the extracted portion of the original binary, leaving only the stub GPU code in the GPU kernel. As shown in Fig. 7, we distribute the GPU program containing the nullified GPU kernel to the host, where the cryptographic operations will be performed. This distribution does not require a secure channel, because it does not distribute any security-sensitive code. When the host calls the OBMI library function, the GPU driver copies the GPU kernel, filled with nullified code, into the device memory and initiates it. Even after nullifying the part of the GPU binary, it can be launched using the CUDA API.

Unlike the nullified GPU binary processed by the above operations, the extracted fragments must be secured. As mentioned above, the authenticated GPU kernel should preserve its integrity for key-protected execution. In addition to integrity, we must also protect confidentiality in order to prevent more sophisticated attack, such as those described in Section 8.3. Thus, we introduce a symmetric master key used to create a secure channel between the SMI handler (i.e., SMRAM in the figure) and the external station. The external station copies the master key into the SMRAM during secure boot sequence. As a result, the extracted fragments can be distributed through the secure channel while preserving both integrity and confidentiality.

When the external station deploys the code fragments into SMRAM, the OBMI library can demand that the SMI handler injects them into the GPU instruction memory. After the nullified GPU kernel is successfully launched by CUDA API, the SMI handler overwrites the instruction memory with the extracted code fragments – i.e., the authenticated GPU code. As previously mentioned, we can pre-calculate the exact location in advance, making it easy to fill the nullified GPU

instruction memory with the extracted code fragments in SMM. The SMI handler injects that portion into the nullified GPU binary using memory-copy operations through MMIO.

The remaining task is to upload the entire GPU kernel with the injected portion into the GPU instruction cache. In order to do so, we simply execute each instruction in the GPU kernel. We can minimize the performance overhead by preventing duplicated executions of the same code. For example, we can minimize duplicated function calls, or enforce loop codes to execute only a single loop. Such optimizations of the code are only adopted during the bootstrapping process. Finally, to preserve the confidentiality of the authenticated GPU code, we remove the code footprint on the device memory before exiting SMM.

7.2. *GPU program verification*

Although we can securely inject the authenticated GPU kernel in SMM, a malicious GPU kernel might nevertheless be launched before the transition to SMM. In such a case, the malicious GPU kernel could bypass the authenticated code injection, and read the secret key as soon as the SMI handler uploads it. It could then write the key into the CPU memory and retrieve it after exiting from SMM. To prevent this from happening, we designed a verification process for code integrity. In other words, we only allow the SMI handler to upload the secret key into the device memory when the SMI handler verifies that the GPU has correctly executed the authenticated GPU kernel. These operations are sequentially described in Fig. 5.

In order to verify the GPU kernel executed by the GPU device, a simple validation number written by the authenticated GPU code is used. The validation number is hardcoded at the beginning of the authenticated GPU code, and it is written into the device memory as soon as the authenticated GPU code initiates. The SMI handler reads the number, and checks its validity. If the number is invalid, the SMI handler does not upload the secret key and terminates the bootstrapping process; otherwise the bootstrapping process continues normally. In either case, the SMI handler removes any footprint of the validation number within the device memory, along with the GPU instructions.

A CPU process cannot acquire this validation number, because no process other than the SMI handler can run in SMM. Therefore, if the above verification step is successfully completed, and the entire authenticated GPU code is uploaded into the instruction cache, only authenticated GPU code will be executed during accelerated cryptographic operations.

8. Evaluation

8.1. *Implementation*

In order to evaluate our mechanism, we developed a prototype running on a commodity CPU and GPU. We implemented our bootstrapping mechanism within the SMI handler. To install our SMI handler, we used coreboot (Coreboot), an open-source BIOS. With coreboot, we can manipulate the BIOS of x86 architectures and unlock the SMRAM region. As a testbed, we used an AMD processor (1.6 GHz, dual-core) and Nvidia GTX 480.

To adopt our mechanism for common GPU cryptographic operations, we developed a CUDA version of RSA and AES based on recent GPU proposals (Harrison and Waldron, 2009; Manavski, 2007). We leveraged the Chinese Remainder Theorem (CRT) and the Constant Length Nonzero Windows (CLNW) partitioning algorithm (Koc, 1995) to reduce the number of exponentiation operations with RSA decryption. In addition, we exploited the Montgomery multiplication technique (Montgomery, 1985) to improve the efficiency of these exponentiation operations. Because these optimization techniques require additional structure, the total size of the key in our RSA-1024 decryption implementation was 8192 bits (i.e., 1024 bytes). For AES, we implemented CBC mode and assumed a 128-bit symmetric key. For security, every round key of AES should be protected, such that the total size of the secret key increases to 176 bytes. We ensured that the GPU executed all RSA and AES operations, and that the CPU merely copied data.

8.2. *Performance evaluation*

We measured the latency incurred by the proposed mechanism, and the results are shown in Table 1. We measured the execution time for each step of our bootstrapping mechanism during the acceleration of RSA and AES. The results shown are the average after 10 runs. The time for SMM transition includes both entering and exiting SMM. We measured this latency by implementing an empty SMI handler that executes a resume instruction (RSM) directly. As the input data for cryptographic operations, we generate 15 requests for small workload and 3920 requests for large workload.

The results show that the additional latency required by our bootstrapping mechanism is relatively small, compared to the overall execution time needed for cryptographic operations. We observed the most significant latency when the GPU cache was filled. However, its latency is still significantly low, compared to the latency from the cryptographic operations. This is due to the optimization techniques we employed, as described in Section 7.1. Even though the same code was executed for both tasks, we efficiently minimize the latency for the security-related step (i.e., filling the GPU cache) by avoiding the duplicated execution of the code.

The copy tasks with the authenticated GPU kernel also incurred some latency for both RSA and AES. Specifically, RSA needed 264.3 μs, which is considerably longer than AES (44.4 μs). This is due to the difference in code size, which was 22.24 KB for RSA and 4.27 KB for AES.

Table 1 – Latency breakdown for bootstrapped cryptographic operations.

Task	Latency in μs	
	RSA	AES
SMM transition	170.8	170.8
GPU code copy	264.3	44.4
GPU code validation	31.0	30.9
Secret key copy	28.8	18.9
Filling GPU cache	2,629.9	43.4
Deletion of sensitive information	35.7	17.8
Cryptographic operations – small workload	56,622.7	103.3
Cryptographic operations – large workload	119,780.6	15,391.7

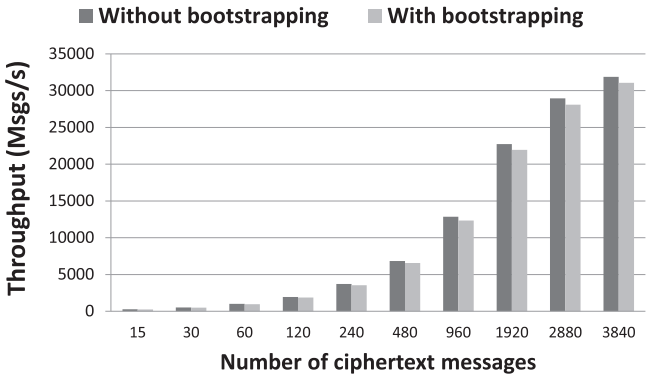

Fig. 8 – Performance comparison for RSA decryption with different numbers of requests.

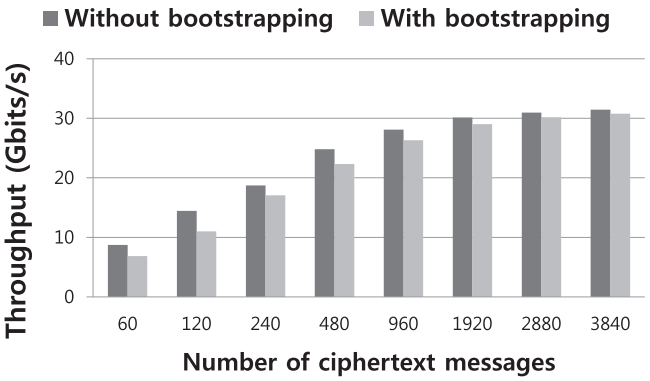

Fig. 9 – Performance comparison for AES decryption with different numbers of requests.

When deleting sensitive information, the SMI handler overwrites the secret key located in the device memory with a zero array. All instructions relating to the GPU code validation number are also overwritten, in order to preserve its confidentiality. Owing to the relatively small size of these structures, they incurred nominal latency.

The percentage of overall latency with our bootstrapping mechanism is also minimal. For large workloads, the overall latency of cryptographic operations increased by 2.9% and 2.7%, for RSA and AES, respectively. This shows that our mechanism is a low-cost solution for securing a GPU. Moreover, our mechanism rarely affects the performance of the host process, because SMM is used for only a short time, while assigning most computations to the GPU.

Figs 8 and 9 show how the throughput changes with our bootstrapping mechanism. We tested various request sizes and compared the throughput with a baseline, labeled "without bootstrapping". The baseline was established by simply accelerating cryptographic operations without any security measures proposed. When calculating the total execution time, we excluded the overhead imposed by data copying in order to measure the precise impact of our mechanism on cryptographic operations. As shown in the figure, our mechanism incurred negligible performance degradation for both RSA and AES. For RSA, the largest degraded throughput showed 95% of the baseline throughput. For large input data, up to 98% of the baseline performance was obtained. As the amount of input data increased, the overhead from bootstrapping diminished, because our bootstrapping mechanism incurs only a fixed amount of latency, and this is independent of input data size. The result of AES is similar to that of RSA, but resulted in a much lower performance for small input data. This was mainly due to the small computational latency of AES itself. AES resulted in up to 24% throughput degradation for a small input size. This implies that frequent requests for bootstrapping with small input data might be ineffective, especially for cryptographic operations requiring small computations.

In Fig. 10, we show how the performance of our bootstrapping mechanism changes when the total key size varies. We compared the overall throughput using multiple symmetric keys for AES decryption. We tested the same number of ciphertexts, 2160 messages, for each experiments. We changed the number of symmetric keys utilized, from 1 to 270. The results show that the overall performance of our mechanism decreased as the number of keys increased. This is mainly because the overhead imposed by copying and deleting the key increased with the number of keys. Moreover, the results of the cryptographic operations became increasingly incorrect when we used more than 270 keys, due to the limited cache capacity. Thus, careful consideration is needed when the multiple secret keys are leveraged.

Finally, Fig. 11 compares the running time of the key-generation process. Our key generation algorithm expanded the random initial vector to generate four 128-bit keys. We

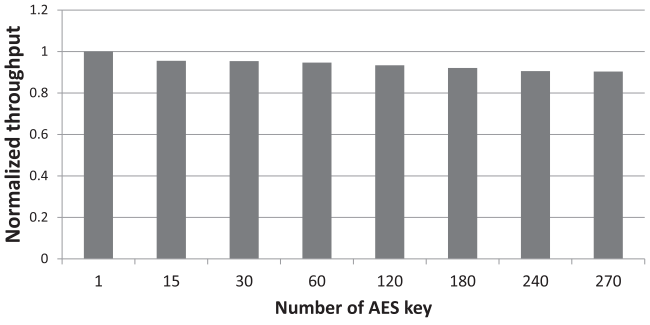

Fig. 10 – Normalized performance for AES decryption with multiple secret keys.

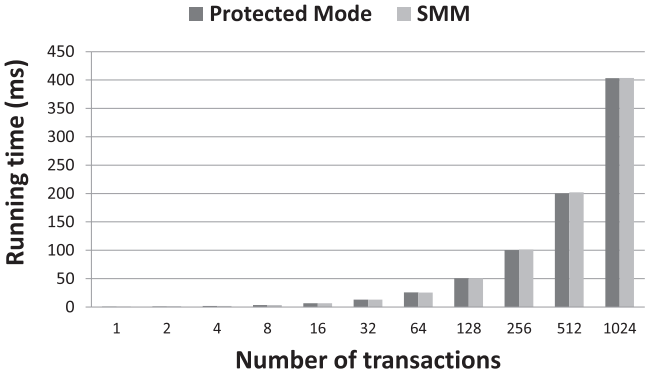

Fig. 11 – Performance comparison of different CPU modes for key-generation process.

implemented a key-expansion mechanism using a pseudo-random function based on SHA256. We used the same algorithm in protected mode and SMM. The results show that SMM does not incur any more performance overhead than the protected mode. As a result, OBMI can implement a safe key-generation mechanism in SMM, enabling the implementation of a full protocol such as SSL, without any performance loss.

8.3. Security analysis

In this section, we discuss several possible attacks on a GPU and how our mechanism protects sensitive information from such attacks.

8.3.1. Modification of the GPU driver or library

When an OS kernel is compromised, attackers can modify the GPU driver or CUDA API to inject malicious code. Moreover, attackers might alter the OBMI library as it is exposed to the user memory space. Although such a maliciously modified library or driver code might be inadvertently executed, this would not result in a security breach. This is because OBMI decouples security-related tasks from the ordinary GPU abstraction layer. We assign only security-unrelated tasks, such as GPU resource allocation, to the ordinary GPU driver or user-level library. Security-related tasks are exclusively handled by the SMI handler, and thus protected within SMRAM.

8.3.2. Malicious GPU kernel

Our bootstrapping mechanism leverages the user-level CUDA driver to launch the nullified GPU kernel, as shown in Fig. 7. Because the nullified GPU kernel is exposed in normal system memory, attackers might replace the GPU kernel with a malicious GPU kernel. In this case, the malicious GPU kernel can obtain any information while it operates within the GPU device, even in SMM. To prevent such malicious actions, we introduced the verification mechanism discussed in Section 7.2. This mechanism uploads the secret key into the GPU device memory only when the GPU kernel has been successfully verified. Moreover, SMRAM cannot be accessed by the GPU kernel, and this prevents a malicious GPU kernel from accessing the secret key.

It is possible for a malicious GPU kernel to attack the verification mechanism directly. That is, a GPU kernel might modify the uploaded authenticated GPU kernel, or even the validation code itself. However, based on our experiments, the GPU kernel cannot modify GPU instruction memory. To the best of our knowledge, memory operations on the code region are impossible for device code. As a result, our authenticated GPU code and verification mechanism are safe from a malicious GPU kernel.

With the Fermi architecture (Wittenbrink et al., 2011), different kernels can be simultaneously launched in order to maximize GPU resource utilization. In this case, a malicious kernel might be executed concurrently with our authenticated GPU code. To prevent this from happening, the GPU block index is checked during the verification process. By checking the total number of validated GPU blocks, we can determine whether all SMs are allocated to authenticated GPU code. Because each SM can only execute a single GPU block, and because there is no way to halt individual GPU blocks, we can guarantee that the entire GPU is dedicated to our authenticated GPU kernel.

8.3.3. GPU code-injection attack

GPU instructions are located within the GPU device memory, to which attackers with kernel privileges have access. With MMIO operations, attackers can inject malicious GPU code that contains commands for accessing the GPU constant cache and writing the secret key to device memory. However, GPU code injection does not affect the bootstrapped GPU execution, because all instructions are uploaded into the GPU instruction cache. We restricted the size of the authenticated GPU kernel to less than the GPU instruction cache, such that the GPU device never references the instruction memory.

However, there might be a way to flush the GPU instruction cache in order to facilitate a GPU code-injection attack. When attackers can flush the GPU instruction cache after injecting the malicious code, the injected code will be executed. Since the GPU device can be controlled by malicious MMIO operations, we investigated MMIO documents from the reverse-engineering community (Envytools; Nouveau) to find cache-related MMIO operations. We have tested every MMIO operation containing "flush" in its name, and none of them flushes the instruction cache. To the best of our knowledge, the only way to flush the GPU instruction cache is to restart a new GPU kernel. In this case, any secret keys in the constant cache will also be flushed.

8.3.4. GPU instruction cache poisoning attack

Although there is no way to directly manipulate the GPU instruction cache, it might be possible to fill the cache with malicious code, using the same process that we used to fill the cache with authenticated code. When a malicious user fills the GPU instruction cache before the bootstrapping process begins, the authenticated GPU code might be compromised.

However, this sophisticated attack can be prevented by improving our verification mechanism (see Section 7.2). One way to do so is simply to hide the entire authenticated GPU code by deleting it before exiting SMM. Because the authenticated GPU code is uploaded in SMM, and because a malicious GPU kernel cannot retrieve the GPU code, as explained above, attackers cannot retrieve any part of the authenticated GPU code. Using this property, we can extend our GPU verification process to check for the results of the authenticated GPU code. As described in Section 7, our bootstrapping mechanism executes the entire authenticated GPU code and fills the cache with it. If we modified the authenticated GPU code to use a random validation number as input data, attackers could not mimic the result of the authenticated GPU code. Consequently, we could immediately detect malicious operations by checking the result of the authenticated GPU code, if an attacker attempts to modify any block of the cache. Because the cache block is 128 bytes in the GPU instruction cache, there is a negligible chance of poisoning the cache without changing the result. Moreover, no additional performance overhead would be incurred by this extended prevention mechanism, because the SMI handler can concurrently execute this supplementary step – i.e., deleting the entire authenticated GPU code – while the GPU accelerates the cryptographic operations.

9. Discussion

9.1. *Deployment process*

To apply our proposal securely and successfully, several components require a secure deployment process. First, the SMI handler must be updated, and this requires either applying a patch to the existing BIOS, or for manufacturers to implement our solution.

As shown in Fig. 7, a secure channel is needed in order to deploy the authenticated GPU code. For this purpose, we utilize the master key, which is located in both the SMI handler and the external station. The master key is established during the secure boot sequence. During the deployment of the GPU code, the confidentiality should be preserved in order to prevent a GPU cache poisoning attack of the kind described in Section 8.3.

The master key is also utilized for encrypting/decrypting the authenticated GPU code. Because of this, we can store the authenticated code within non-volatile storage in encrypted form, and the SMI handler can decrypt its content whenever the system reboots. As a result, the GPU code must be deployed only once using a secure channel.

9.2. *Unexpected cache behavior*

When implementing OBMI, unexpected cache behavior should be monitored, because OBMI leverages the GPU cache to store the GPU code and secret keys. Because the cache is a shared resource with a limited capacity, the secret key might be overwritten by other data accessing the cache. We avoid this problem by stipulating that the secret key uses constant memory exclusively. However, because the L2 and L3 constant memory caches are shared with GPU instructions (Wong et al., 2010), the large size of the GPU code can result in an unexpected cache interference. In this case, either we should use fewer secret keys, or we should reduce the size of GPU code. For example, we can significantly reduce the code size by disabling an inline function.

This kind of code minimization is also needed when leveraging our solution for complex cryptographic operations. In the case of RSA, a nave implementation results in a large GPU code size (81.67 KB), which exceeds the capacity of the GPU instruction cache. Fortunately, the GPU code is reduced to 22.24 KB after disabling the inline function. To prevent performance degradation, we carefully selected a function to disable the inline property.

Prefetching is unexpected cache behavior that should also be considered. Because the GPU instruction cache supports an instruction prefetching mechanism, our injection mechanism can be affected in unexpected ways. Specifically, as the stub GPU code awaits the injection of the authenticated GPU code, a few bytes of nullified instructions can be prefetched in advance in lieu of the authenticated GPU code. To prevent this prefetching malfunction, we inserted a few dummy instructions ahead of the authenticated GPU code.

9.3. *Unrevealed GPU hardware mechanism*

Although we analyzed various attacks in the previous section, other GPU attacks are possible based on unrevealed hardware features in the GPU device. For example, a cache flush attack can be leveraged with GPU MMIO, as discussed in Section 8.3, even though we confirmed that such attacks are currently non-existent. Similar attacks are nevertheless possible, with unrevealed MMIO or GPU hardware mechanisms. To investigate the feasibility of such attacks, we encourage researchers to actively analyze the internal GPU hardware and control mechanisms.

9.4. *Portability*

Although our solution is based on the characteristics of GPU hardware, we exclusively utilized general structures, such as the instruction cache or constant cache. Dedicated constant memory is utilized for all CUDA-enabled Nvidia GPUs, and a dedicated instruction cache is also common. Moreover, our injection mechanism is based on the fact that the GPU statically utilizes the physical memory for running the same program. We believe that this is generally true for all GPU architectures, because all CUDA-enabled GPUs have a so-called non-preemptiveness property that simplifies memory management.

10. Conclusion

In this paper, we presented the design and implementation of OBMI, a framework for bootstrapping secure cryptographic operations on commodity GPUs. By utilizing SMM as an isolated execution environment, we developed an on-demand mechanism that securely establishes authenticated GPU binaries and secret keys in GPU caches. Our mechanism enables the secure acceleration of various cryptographic operations using commodity hardware features. OBMI is transparent to any system software, whether an OS, hypervisor, or GPU device driver. Moreover, the security offered by OBMI does not depend on the integrity of such system software. Our evaluations showed that OBMI incurred only a negligible performance overhead for both RSA and AES. We also demonstrated that even privileged malicious software could not retrieve sensitive information from OBMI's bootstrapped GPU computations.

In future work, we plan to extend our OBMI framework to provide more sophisticated tasks, such as HTTP transactions. To do so, we will explore balanced CPU/GPU utilization for intensive server workloads, while preserving security even in a compromised OS kernel. It is also possible to apply our mechanism to different system architectures – for example, to mobile devices, most of whose platforms are based on ARM, where the x86-based SMM is unavailable.

Acknowledgement

This work was supported by the ICT R&D program of MSIP/IITP (R0126-16-1005, Development of High Reliable Communications and Security SW for Various Unmanned Vehicles) and Institute for Information & Communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (No. R0190-15-2010, Development on the SW/

HW modules of Processor Monitor for System Intrusion Detection).

REFERENCES

Abe Y, Sasaki H, Peres M, Inoue K, Murakami K, Kato S. Power and performance analysis of gpu-accelerated systems. In: Proceedings of the 2012 USENIX conference on power-aware computing and systems, HotPower'12. Berkeley, CA, USA: USENIX Association; 2012. p. 10. <http://dl.acm.org/citation.cfm?id=2387869.2387879>.

Advanced Micro Devices. Amd64 virtualization: secure virtual machine architecture reference manual. May 2005.

Advanced Micro Devices. Amd64 architecture programmer's manual: volume 2: system programming. 2013. <http://developer.amd.com/wordpress/media/2012/10/24593_APM_v21.pdf>.

ARM. ARM security technology – building a secure system using trustzone technology. ARM technical white paper. 2009. <http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf>.

Azab AM, Ning P, Wang Z, Jiang X, Zhang X, Skalsky NC. Hypersentry: enabling stealthy in-context measurement of hypervisor integrity. In: Proceedings of the 17th ACM conference on computer and communications security, CCS '10. New York, NY, USA: ACM; 2010. p. 38–49. doi:10.1145/1866307.1866313. <http://doi.acm.org/10.1145/1866307.1866313>.

Blass E-O, Robertson W. Tresor-hunt: attacking CPU-bound encryption. In: Proceedings of the 28th annual computer security applications conference, ACSAC '12. New York, NY, USA: ACM; 2012. p. 71–8. doi:10.1145/2420950.2420961. <http://doi.acm.org/10.1145/2420950.2420961>.

Cheng J, Grossman M, McKercher T. Professional CUDA C programming. John Wiley & Sons; 2014.

Coreboot. Coreboot project home page, <http://www.coreboot.org>; [accessed 07.07.16].

CVE Details. Xen: Vulnerability Statistics. CVE Details, <https://www.cvedetails.com/vendor/6276/XEN.html>; [accessed 07.07.16].

Envytools. Envytools project home page, <https://github.com/envytools/envytools>; [accessed 07.07.16].

Falcon Security. NVIDIA Falcon Security, <ftp://download.nvidia.com/open-gpu-doc/Falcon-Security/1/Falcon-Security.html>; [accessed 07.07.16].

Fujii Y, Azumi T, Nishio N, Kato S, Edahiro M. Data transfer matters for GPU computing. In: 19th IEEE international conference on parallel and distributed systems, ICPADS 2013, Seoul, Korea, December 15–18, 2013. IEEE Computer Society; 2013. p. 275–82. doi:10.1109/ICPADS.2013.47. <http://dx.doi.org/10.1109/ICPADS.2013.47>.

Garmany B, Muller T. Prime: private rsa infrastructure for memory-less encryption. In: Proceedings of the 29th annual computer security applications conference, ACSAC '13. New York, NY, USA: ACM; 2013. p. 149–58. doi:10.1145/2523649.2523656. <http://doi.acm.org/10.1145/2523649.2523656>.

Guan L. Protecting private keys against memory disclosure attacks using hardware transactional memory. In: IEEE symposium on security and privacy (SP). San Jose, CA: IEEE; 2015. p. 3–19. <http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=7163015>.

Guan L, Lin J, Luo B, Jing J. Copker: computing with private keys without RAM. In: 21st annual network and distributed system security symposium, NDSS 2014, San Diego, California, USA, February 23–26, 2014. The Internet Society; 2014. <http://www.internetsociety.org/doc/copker-computing-private-keys-without-ram>.

Halderman JA, Schoen SD, Heninger N, Clarkson W, Paul W, Calandrino JA, et al. Lest we remember: cold-boot attacks on encryption keys. Commun ACM 2009;52(5):91–8. doi:10.1145/1506409.1506429. <http://doi.acm.org/10.1145/1506409.1506429>.

Harrison O, Waldron J. Efficient acceleration of asymmetric cryptography on graphics hardware. In: Proceedings of progress in cryptology – AFRICACRYPT 2009: second international conference on cryptology in Africa, Gammarth, Tunisia, June 21–25, 2009. Berlin, Heidelberg: Springer Berlin Heidelberg; 2009. p. 350–67. doi:10.1007/978-3-642-02384-2_22. <http://dx.doi.org/10.1007/978-3-642-02384-2_22>.

Heartbleed. The heartbleed bug, <http://heartbleed.com/>; [accessed 07.07.16].

Hofmann OS, Kim S, Dunn AM, Lee MZ, Witchel E. InkTag: secure applications on an untrusted operating system. SIGPLAN Not. 2013;48(4):265–78. doi:10.1145/2499368.2451146. <http://doi.acm.org/10.1145/2499368.2451146>.

Intel Corporation. Intel trusted execution technology. 2013. <http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/trusted-execution-technology-security-paper.pdf>.

Intel Corporation. Software guard extensions programming reference. October 2014. <https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf>.

Intel Corporation. Intel 64 and ia-32 architectures software developer's manual vol. 3a:system programming guide. 2016. <http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.html>.

Jang K, Han S, Han S, Moon S, Park K. SSLShader: cheap SSL acceleration with commodity processors. In: Proceedings of the 8th USENIX conference on networked systems design and implementation, NSDI'11. Berkeley, CA, USA: USENIX Association; 2011. p. 1–14. <http://dl.acm.org/citation.cfm?id=1972457.1972459>.

Koc CK. Analysis of sliding window techniques for exponentiation. Comput Math Appl 1995;30:17–24.

Larabel M. Benchmarking Nouveau and NVIDIA's proprietary GeForce driver on Linux. 2014. <http://www.phoronix.com/scan.php?page=article&item=nvidia_nouveau_linux316&num=1> [last accessed 17.02.16].

Lee MS, Lee Y, Cheon JH, Paek Y. Accelerating bootstrapping in FHEW using GPUs. In: ASAP. IEEE; 2015. p. 128–35. <http://dblp.uni-trier.de/db/conf/asap/asap2015.html#LeeLCP15>.

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 2014, Berkeley, CA, USA, May 18–21, 2014. IEEE Computer Society; 2014. p. 19–33. doi:10.1109/SP.2014.9. <http://dx.doi.org/10.1109/SP.2014.9>.

Liu M, Li T, Jia N, Currid A, Troy V. Understanding the virtualization "tax" of scale-out pass-through GPUs in GaaS clouds: an empirical study. In: HPCA. IEEE; 2015. p. 259–70. <http://dblp.uni-trier.de/db/conf/hpca/hpca2015.html#LiuLJCT15>.

Manavski SA. CUDA compatible GPU as an efficient hardware accelerator for AES cryptography. In: IEEE international conference on signal processing and communications. ICSPC 2007. Dubai: IEEE; 2007. p. 65–8. <http://ieeexplore.ieee.org/xpls/icp.jsp?arnumber=4728256>.

Maurice C, Neumann C, Heen O, Francillon A. Confidentiality issues on a GPU in a virtualized environment. In: FC 2014, 18th international conference on financial cryptography and data security, 3–7 March 2014. Barbados: 2014. doi:10.1007/

978-3-662-45472-5_9. <http://www.eurecom.fr/publication/4205>.
McCune JM, Parno BJ, Perrig A, Reiter MK, Isozaki H. Flicker: an execution infrastructure for TCB minimization. In: Proceedings of the 3rd ACM SIGOPS/EuroSys European conference on computer systems 2008, Eurosys '08. New York, NY, USA: ACM; 2008. p. 315–28. doi:10.1145/1352592.1352625. <http://doi.acm.org/10.1145/1352592.1352625>.
McCune JM, Li Y, Qu N, Zhou Z, Datta A, Gligor V, et al. Trustvisor: efficient TCB reduction and attestation. In: Proceedings of the 2010 IEEE symposium on security and privacy, SP '10.Washington, DC, USA: IEEE Computer Society; 2010. p. 143–58. doi:10.1109/SP.2010.17. <http://dx.doi.org/10.1109/SP.2010.17>.
Micikevicius P. Local memory and register spilling. 2011. <http://on-demand.gputechconf.com/gtc-express/2011/presentations/register_spilling.pdf> [last accessed 17.02.16].
Miele A. Buffer overflow vulnerabilities in CUDA: a preliminary analysis. CoRR abs/1506.08546. 2015. <http://arxiv.org/abs/1506.08546>.
Montgomery PL. Modular multiplication without trial division. Math Comput 1985;44(170):519–21.
Müller T, Dewald A, Freiling FC. Aesse: a cold-boot resistant implementation of aes. In: Proceedings of the third European workshop on system security, EUROSEC '10. New York, NY, USA: ACM; 2010. p. 42–7. doi:10.1145/1752046.1752053. <http://doi.acm.org/10.1145/1752046.1752053>.
Müller T, Freiling FC, Dewald A. Tresor runs encryption securely outside ram. In: Proceedings of the 20th USENIX conference on security, SEC'11. Berkeley, CA, USA: USENIX Association; 2011. p. 17. <http://dl.acm.org/citation.cfm?id=2028067.2028084>.
Nouveau. Nouveau project home page, <http://nouveau.freedesktop.org>; [accessed 07.07.16].
Nvidia Corporation. CUDA C programming guide v7.5. Online. September 2015. <http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf> [last accessed 17.02.16].
Pietro RD, Lombardi F, Villani A. CUDA leaks: a detailed hack for CUDA and a (partial) fix. ACM Trans Embed Comput Syst 2016;15(1):15:1–25. doi:10.1145/2801153. <http://doi.acm.org/10.1145/2801153>.
Sani AA, Zhong L, Wallach DS. Glider: a GPU library driver for improved system security. 2014. CoRR abs/1411.3777. <http://arxiv.org/abs/1411.3777>.
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, ACSAC '11. New York, NY, USA: ACM; 2011. p. 73–82. doi:10.1145/2076732.2076743. <http://doi.acm.org/10.1145/2076732.2076743>.
Suzuki Y, Kato S, Yamada H, Kono K. GPUvm: why not virtualizing GPUs at the hypervisor? In: Gibson G, Zeldovich N, editors. 2014 USENIX annual technical conference, USENIX ATC '14, Philadelphia, PA, USA, June 19–20, 2014. USENIX Association; 2014. p. 109–20. <https://www.usenix.org/conference/atc14/technical-sessions/presentation/suzuki>.
Tian K, Dong Y, Cowperthwaite D. A full gpu virtualization solution with mediated pass-through. In: Proceedings of the 2014 USENIX conference on USENIX annual technical conference, USENIX ATC'14. Berkeley, CA, USA: USENIX Association; 2014. p. 121–32. <http://dl.acm.org/citation.cfm?id=2643634.2643647>.
Vasiliadis G, Athanasopoulos E, Polychronakis M, Ioannidis S. Pixelvault: using GPUs for securing cryptographic operations. In: Ahn G, Yung M, Li N, editors. Proceedings of the 2014 ACM SIGSAC conference on computer and communications security, Scottsdale, AZ, USA, November 3–7, 2014. ACM; 2014. p. 1131–42. doi:10.1145/2660267.2660316. <http://doi.acm.org/10.1145/2660267.2660316>.
Wang W, Chen Z, Huang X. Accelerating leveled fully homomorphic encryption using GPU. In: IEEE international symposium on circuits and systems, ISCAS 2014, Melbourne, Victoria, Australia, June 1–5, 2014. IEEE; 2014. p. 2800–3. doi:10.1109/ISCAS.2014.6865755. <http://dx.doi.org/10.1109/ISCAS.2014.6865755>.
Wittenbrink CM, Kilgariff E, Prabhu A. Fermi gf100 gpu architecture. IEEE Micro 2011;31(2):50–9. <http://dblp.uni-trier.de/db/journals/micro/micro31.html#WittenbrinkKP11>.
Wong H, Papadopoulou M, Sadooghi-Alvandi M, Moshovos A. Demystifying GPU microarchitecture through microbenchmarking. In: IEEE international symposium on performance analysis of systems and software, ISPASS 2010, 28–30 March 2010. White Plains, NY, USA: IEEE Computer Society; 2010. p. 235–46. doi:10.1109/ISPASS.2010.5452013. <http://dx.doi.org/10.1109/ISPASS.2010.5452013>.
Yu M, Gligor VD, Zhou Z. Trusted display on untrusted commodity platforms. In: Proceedings of the 22nd ACM SIGSAC conference on computer and communications security, CCS '15. New York, NY, USA: ACM; 2015. p. 989–1003. doi:10.1145/2810103.2813719. <http://doi.acm.org/10.1145/2810103.2813719>.
Zaddach J, Kurmus A, Balzarotti D, Blass E-O, Francillon A, Goodspeed T, et al. Implementation and implications of a stealth hard-drive backdoor. In: Proceedings of the 29th annual computer security applications conference, ACSAC '13. New York, NY, USA: ACM; 2013. p. 279–88. doi:10.1145/2523649.2523661. <http://doi.acm.org/10.1145/2523649.2523661>.
Zhang F, Leach K, Sun K, Stavrou A. SPECTRE: a dependable introspection framework via system management mode. In: 2013 43rd annual IEEE/IFIP international conference on dependable systems and networks (DSN), Budapest, Hungary, June 24–27, 2013. IEEE; 2013. p. 1–12. doi:10.1109/DSN.2013.6575343. <http://doi.ieeecomputersociety.org/10.1109/DSN.2013.6575343>.
Zhang F, Wang H, Leach K, Stavrou A. A framework to secure peripherals at runtime. In: Kutylowski M, Vaidya J, editors. Computer security – ESORICS 2014 – 19th European symposium on research in computer security, Wroclaw, Poland, September 7–11, 2014. Proceedings, part I, vol. 8712. Lecture Notes in Computer Science. Springer; 2014. p. 219–38. doi:10.1007/978-3-319-11203-9_13. <http://dx.doi.org/10.1007/978-3-319-11203-9_13>.
Zhang F, Wang J, Sun K, Stavrou A. Hypercheck: a hardware-assisted integrity monitor. IEEE Trans Dependable Secure Comput 2014;11(4):332–44. doi:10.1109/TDSC.2013.53. <http://doi.ieeecomputersociety.org/10.1109/TDSC.2013.53>.
Zhang F, Leach K, Stavrou A, Wang H, Sun K. Using hardware features for increased debugging transparency. In: 2015 IEEE symposium on security and privacy, SP 2015, San Jose, CA, USA, May 17–21, 2015. IEEE Computer Society; 2015. p. 55–69. doi:10.1109/SP.2015.11. <http://dx.doi.org/10.1109/SP.2015.11>.
Zheng F, Pan W, Lin J, Jing J, Zhao Y. Exploiting the floating-point computing power of gpus for RSA. In: Chow SSM, Camenisch J, Hui LCK, Yiu S, editors. Proceedings of information security – 17th international conference, ISC 2014, Hong Kong, China, October 12–14, 2014, vol. 8783. Lecture Notes in Computer Science. Springer; 2014. p. 198–215. doi:10.1007/978-3-319-13257-0_12. <http://dx.doi.org/10.1007/978-3-319-13257-0_12>.

Yonggon Kim is a Ph.D. candidate in Computer Science at Korea Advanced Institute of Science and Technology (KAIST). He received his B.S. degree in Computer Science from KAIST in

2008. He also received his M.S. degree of Computer Science from KAIST in 2011. His research interests include graphics hardware and computer security, especially in hardware-assisted security.

Ohmin Kwon received the B.S degree in Division of Computer and Communication Engineering from Korea University, South Korea, in 2012. He also received the M.S. degree in Computer Science from Korea Advanced Institute of Science and Technology (KAIST) in 2014. He is currently working toward the Ph.D. degree at the Division of Computer Science, KAIST. His research interests include computer security and applied cryptography.

Jinsoo Jang received the B.E. degree in Information and Computer Engineering from Ajou University, South Korea, in 2007. He also received the M.S. degree in Information Security from Korea Advanced of Science and Technology (KAIST) in 2014. He is currently working toward the Ph.D. degree at the Division of Computer Science, KAIST. His research interest includes system security, especially in the trusted execution environments (TEE).

Seongwook Jin is a Ph.D. candidate in Computer Science at Korea Advanced Institute of Science and Technology (KAIST). His research interests are in computer architecture, security and virtualization. He received the B.S. degree in Computer Science and Engineering from Kumoh National University in 2008 and the M.S. degree in Computer Science from KAIST in 2010.

Hyeongboo Baek is a Ph.D. student at the Department of Computer Science at Korea Advanced Institute of Science and Technology (KAIST), South Korea. He received B.S. degree in Computer Science and Engineering from Konkuk University, South Korea, in 2010 and M.S. degree in Computer Science from KAIST, South Korea, in 2012. His research interests include real-time embedded systems, cyber-physical systems and security.

Brent Byunghoon Kang is currently an associate professor at the GSIS (Graduate School of Information Security) at KAIST (Korea Advanced Institute of Science and Technology). Before KAIST, he has been with George Mason University as an associate professor in the Volgenau School of Engineering. Dr. Kang received his Ph.D. in Computer Science from the University of California at Berkeley, and M.S. from the University of Maryland at College Park, and B.S. from Seoul National University. He has been working on systems security area including OS kernel integrity monitor, trusted execution environment, hardware-assisted security, botnet malware defense, and DNS analytics.

Hyunsoo Yoon received the B.E. degree in electronics engineering from Seoul National University, South Korea, in 1979, the M.S. degree in computer science from Korea Advanced Institute of Science and Technology (KAIST) in 1981, and the Ph.D. degree in computer and information science from the Ohio State University, Columbus, Ohio, in 1988. From 1988 to 1989, he was a member of technical staff at AT&T Bell Labs. Since 1989 he has been a faculty member of Division of Computer Science at KAIST. His main research interest includes wireless sensor networks, 4G networks, and network security.