Confidentiality Issues on a GPU in a Virtualized Environment

Clémentine Maurice[1,2], Christoph Neumann[1], Olivier Heen[1],
Aurélien Francillon[2]

[1] Technicolor, Rennes, France
[2] Eurecom, Sophia Antipolis, France

Abstract. General-Purpose computing on Graphics Processing Units (GPGPU) combined to cloud computing is already a commercial success. However, there is little literature that investigates its security implications. Our objective is to highlight possible information leakage due to GPUs in virtualized and cloud computing environments. We provide insight into the different GPU virtualization techniques, along with their security implications. We systematically experiment and analyze the behavior of GPU global memory in the case of direct device assignment. We find that the GPU global memory is zeroed only in some configurations. In those configurations, it happens as a side effect of Error Correction Codes (ECC) and not for security reasons. As a consequence, an adversary can recover data of a previously executed GPGPU application in a variety of situations. These situations include setups where the adversary launches a virtual machine after the victim's virtual machine using the same GPU, thus bypassing the isolation mechanisms of virtualization. Memory cleaning is not implemented by the GPU card itself and we cannot generally exclude the existence of data leakage in cloud computing environments. We finally discuss possible countermeasures for current GPU clouds users and providers.

Keywords: GPU, Security, Cloud Computing, Information Leakage

1 Introduction

Graphics Processing Units (GPUs) benefit from a great interest from the scientific community since the rise of General-Purpose computing on Graphics Processing Units (GPGPU) programming. GPGPU allows performing massively parallel general purpose computations on a GPU by leveraging the inherent parallelism of GPUs. GPUs exploit hundreds to thousands of cores to accelerate parallel computing tasks, such as financial applications [8,22,40], encryption [16,45], and Bitcoin mining [23]. They are also used as a co-processor to execute malicious code that evades detection [24,41], or on the opposite to monitor security [26]. GPUs have recently been offered by several cloud computing providers to supply on demand and pay-per-use of otherwise very expensive hardware.

While GPU Clouds have been mainly used for on demand high performance computing, other applications emerge. For example, in *cloud gaming* game rendering is done in the cloud allowing to play to GPU intensive games on low end devices, such as tablets. Virtualized workstations allow performing data and graphically intensive tasks on regular desktops or laptops, such as movie editing or high-end computer aided design.

GPUs have been designed to provide maximum performance and throughput. They have not been designed for concurrent accesses, that is to support virtualization or simultaneous users that share the same physical resource. It is known that GPU buffers are not zeroed when allocated [20]. This raises confidentiality issues between different programs or different users when GPUs are used natively on personal computers [12]. Clearly, the attack surface is larger in a cloud environment when several users exploit the same GPU one after another or even simultaneously. However, such a setup has not been previously studied.

Our objective is to evaluate the security of GPUs in the context of virtualized and cloud computing environments, and particularly to highlight potential information leakage from one user to another. This is a topic of interest since users cannot trust each other in the cloud environment. However, identifying possible information leakage in such environments is an intricate problem since we are faced with two layers of obscurity: the cloud provider as well as the GPU.

Contributions

In this paper, we study information leakage on GPUs and evaluate its possible impact on GPU clouds. We systematically experiment and analyze the behavior of GPU global memory in non-virtualized and virtualized environments. In particular:

1. We give an overview of existing GPU virtualization techniques and discuss the security implications for each technique.
2. We reproduce and extend recent information leakage experiments on non-virtualized GPUs [9,12]. In addition to previous work, we show how an adversary can retrieve information from GPU global memory using a variety of proprietary and open-source drivers and frameworks. Furthermore, we find that in the rare cases where GPU global memory is zeroed, it is only as a side effect of Error Correction Codes (ECC) and not for security reasons. We also propose a method to retrieve memory in a driver agnostic way that bypasses some memory cleanup measures a conscious programmer may have implemented.
3. We experiment the case of virtual environments with lab testbeds under Xen and KVM using a GPU in direct device assignment mode, which is the GPU virtualization technique most commonly used in GPU clouds. We also conduct experiments on a real life cloud. We explain under which conditions and how an adversary can retrieve data from GPU global memory of an application that has been executed on a different virtual machine (VM).
4. We present recommendations to limit information leakage in cloud and virtualized environments.

The remainder of this paper is organized as follows. Section 2 presents the background related to GPUs and the related work on information leakage and GPU virtualization. Section 3 details our adversary model and the security impact of the different GPU virtualization techniques. Section 4 exposes our experiments, organized according to two main parameters: the degree of virtualization and the method used to access the memory. Section 5 details the experiments that leverage GPGPU runtime to access the memory, and Section 6 the experiments that exploit the PCI configuration space. Section 7 presents possible countermeasures. Section 8 concludes.

2 Background

In this section, we recall basic notions on GPUs, as well as related work on information leakage and GPU virtualization.

2.1 GPU Basics

In this paper we focus on NVIDIA GPUs because they are the most widespread devices used in GPGPU applications, yet they are poorly documented. The Tesla architecture[3] introduced a general purpose pipeline, followed by the Fermi and, the latest, Kepler architecture. GPUs handle throughput-based workloads that have a large degree of data parallelism. GPUs have hundreds of cores that can handle hundreds of threads to mitigate the latency caused by the limited memory bandwidth and the deep pipeline. A GPU is first composed of several Streaming Multiprocessors (SM), which are in turn composed of Streaming Processor cores (SP, or CUDA cores). The number of SMs depends on the card, and the number of SP per SM depends on the architecture. The Fermi architecture introduces a memory hierarchy. It offers an off-chip DRAM memory and an off-chip L2 cache shared by all SMs. On-chip, each SM has its own set of registers and its own memory partitioned between a L1 cache and a shared memory accessible by the threads running on the SPs. Figure 1 depicts a typical GPU architecture.

CUDA is the most used GPGPU platform and programming model for NVIDIA GPUs. CUDA allows developers to write GPGPU-specific C functions called *kernels*. Kernels are executed n times in parallel by n threads. Each SP handles one or more threads. A group of threads is called a block, and each SM handles one or more blocks. A group of blocks is called a *grid*, and an entire grid is handled by a single GPU. CUDA introduces a set of memory types. *Global*, *texture* and *constant* memories are accessible by all threads of a *grid* and stored on the GPU DRAM. *Local* memory is specific to a thread but stored on the GPU DRAM. *Shared* memory is shared by all threads of a block and stored in shared memory. Finally, registers are specific to a thread and stored on-chip.

[3] Tesla is used by NVIDIA both as an architecture code name and a product range name [25]. NVIDIA commercialized the Tesla architecture under the name GeForce 8 Series. When not specified, we refer to the product range name in the remainder of the article.

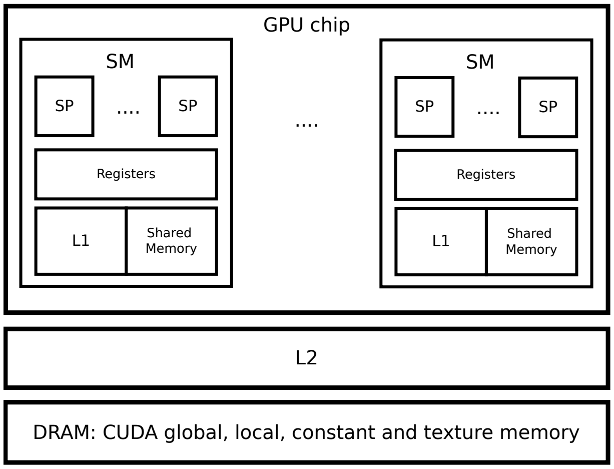

Fig. 1: GPU card with Fermi architecture.

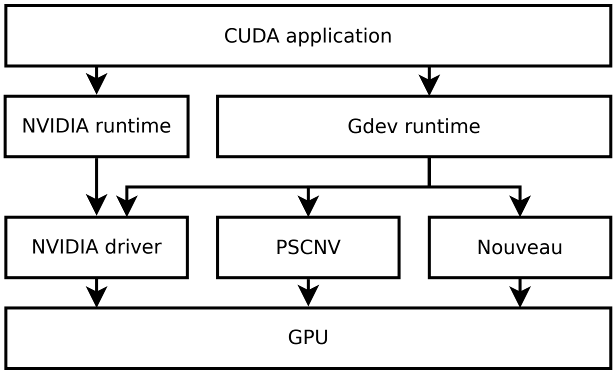

Fig. 2: GPGPU frameworks and their stack.

CUDA programs either run on top of the closed source NVIDIA CUDA runtime or on top of the open-source Gdev [19] runtime. The NVIDIA CUDA runtime relies on the closed-source kernel-space NVIDIA driver and a closed-source user-space library. Gdev supports the open source *Nouveau* driver [28], the PSCNV driver [33] and the NVIDIA driver. Both closed-source and open-source solutions support the same APIs: CUDA programs can be written using the runtime API, or the driver API for low-level interaction with the hardware [30]. Figure 2 illustrates the stack of CUDA and Gdev frameworks under Linux.

2.2 Information Leakage

Information Leakage in Cloud Computing Information leakage in cloud computing has been extensively studied and related work mainly focus on deduplication, side and covert channels. Harnik et al. [18] show the implications of file-level deduplication in terms of covert and side channels; Suzaki et al. [39] reveal that page-level deduplication can be used to infer the applications that are running on other VMs; Owens et al. [32] infer the OS of other VMs using deduplication. Ristenpart et al. [35] study the extraction of confidential information via a coarse grained side channel on the data cache. Zhang et al. [46] exploit a side channel on the L1 (CPU) instruction cache across VMs. Wu et al. [44] assert that cache covert and side channels are not practical due to the distribution of virtual cores among physical cores. They propose a new bus-contention based covert channel, that uses atomic instructions to lock the shared memory bus.

Information Leakage in GPUs Using the CUDA framework, Di Pietro et al. [12] show that GPU architectures are vulnerable to information leakage, mainly due to memory isolation issues. The leakage affects the different memory spaces in GPU: global memory, shared memory, and registers. Di Pietro et al. also show that current implementations of AES cipher that leverage GPUs allow recovering both plaintext and encryption key in the GPU global memory. Bress et al. [9] consider using these vulnerabilities to perform forensic investigations. Nevertheless, they note that we cannot guarantee that calls to the CUDA API do not modify the memory. These two works begin to pave the way of GPU security, however they do not evaluate information leakage by GPUs in the context of virtualization that is characteristic of cloud computing.

2.3 GPU Virtualization

In virtualized environments, guest VMs are running isolated from each other and managed by a privileged VM, while an hypervisor handles access to physical resources. Hardware-assisted virtualization (HVM) was introduced by Intel in VT-x Virtualization Technology (and similarly by AMD in AMD-V) to overcome the performance overhead of software virtualization of the x86 architecture. Examples of commodity hypervisors include Xen and KVM, both of them supporting HVM. KVM is implemented as a kernel-space device driver. Xen is a bare-metal hypervisor, meaning that it runs directly on the host's hardware. At startup, Xen starts the privileged domain that is called Domain-0 (or dom0). The other unprivileged domains are named domU.

Dowty et al. [13] classify GPU virtualization into frontend and backend virtualization. Frontend virtualization puts the virtualization boundary at the host or hypervisor level so that guests only interact with the GPU through software. Solutions go on a continuum between *device emulation* and a *split driver model*, also called *API remoting*. Backend virtualization is also called *direct device assignment* or *PCI passthrough* (both are equivalent). In their performance evaluation, Vinaya et al. [42] concluded that direct device assignment mode is the one that provides the best performance and fidelity.

Emulation When a GPU is emulated, the hypervisor implements in software the features of existing, standard devices – regardless of the actual physical devices. Device emulation does not require any change in the guest OS, which uses standard device drivers. Emulation comes with non negligible overhead, and is therefore not an option for GPUs that are used for high performance computing. The closest approach to full GPU emulation is the one presented by Dowty et al. in [13], which also includes characteristics of API remoting.

Split Driver Model The split driver model, also known as *driver paravirtualization*, involves sharing a physical GPU. Hardware management is left to a privileged domain. A frontend driver runs in the unprivileged VM and forwards calls to the backend driver in the privileged domain. The backend driver then takes care of sharing resources among virtual machines. This approach requires special drivers for the guest VM. In the literature, the methods that use this model virtualize the GPU at the CUDA API level [17,36,15], *i.e.,* the backend drivers in the privileged domain comprise the NVIDIA GPU drivers and the CUDA library. The split driver model is currently the only GPU virtualization technique that effectively allows sharing the same GPU hardware between several VMs *simultaneously* [34,7].

Direct Device Assignment In direct device assignment, the guest VM has direct control on the PCI device. Direct device assignment does not allow several VMs to share the same GPU at the same time, and for the whole duration of the VM. However, it allows several VMs to share the same GPU one after another. Direct device assignment is the most commonly used GPU virtualization mode and it is also used by GPU cloud providers such as Amazon Web Services. To assign a device to a virtual machine, the hypervisor allows the VM to directly access the device's PCI range. A hardware I/O Memory Management Unit (IOMMU), such as Intel's VT-d, thwarts Direct Memory Access (DMA) attacks by preventing devices from accessing arbitrary parts of the physical memory.

Direct Device Assignment with SR-IOV Single Root I/O Virtualization (SR-IOV) capable devices can expose themselves to the operating system as several devices. The hardware device itself can be composed of several independent functions (multiple devices) or multiplex the resources in hardware. This technique therefore provides increased performance. In SR-IOV, the hypervisor controls the assignment of each of the devices to a different guest VM. All isolation mechanisms are implemented in hardware. This technology allows to simultaneously share the same GPU among several tenants. NVIDA only very recently introduced this type of technology as GRID VGX [31], however, we are not aware of any deployment of SR-IOV GPUs by cloud providers.

3 The Security of GPUs in Virtualized Environments

In this section, we present our adversary model, as well as a study of the security of the different GPU virtualization techniques, in terms of information leakage.

3.1 Adversary Model

The objective of the adversary is to learn some information about the victim. This can occur directly by retrieving data owned by the victim in the memory of the GPU, or indirectly through side channels. We assume that the adversary has full control over a VM. In our case, the VM has access to a virtualized GPU. We consider two cases:

- The *serial adversary* has access to the same GPU as the victim's, before or after the victim. She will seek for traces of data previously left in different memories of the GPU. Our experiments, in Section 4 and following, consider this particular adversary.
- The *parallel adversary* and the victim are running simultaneously on the same virtualized GPU. She may also have direct access to memory used by the victim, if memory management is not properly implemented. However, as the *parallel adversary* shares the device with the victim, she may also abuse some side channels on the GPU, possibly allowing her to recover useful information.

The serial adversary can have access to the GPU memory in two different ways. In our experiments, we outline two types of attacks that require different capabilities for the adversary and differ in their results:

- In the first scenario, the adversary accesses portions of the GPU memory through a GPGPU runtime. She does not need root privileges since she uses perfectly legitimate calls to the CUDA runtime API.
- In the second scenario, the adversary accesses the GPU memory through the PCI configuration space; we assume the adversary has root privileges, either because she controls the machine or because she compromised it by exploiting a known privilege escalation. This attack calls for a more powerful adversary, but gives a complete snapshot of the GPU memory.

3.2 GPU Virtualization Technologies Impact on Security

Emulation Emulation is conceptually the safest virtualization technique. This virtualization technique is the one that brings the most interposition, *i.e.,* the hypervisor is able to inspect, and possibly modify or deny, all guests calls. Emulation also implements a narrow API, which limits the attack surface. Emulation often does not rely on actual hardware. Therefore, information leakage – or side channels – that is due to hardware sharing is effectively eliminated.

Split Driver Model The split driver model is prone to information leakage and side channels enabled by the shared hardware. Furthermore, the backend driver has to ensure the isolation of guests that share the same hardware. GPU drivers have not been designed with that goal in mind, therefore, the backend driver should completely be redesigned to address this. From an isolation, interposition and attack surface perspective, the split driver model is somewhere between emulation and direct device assignment. The API exposed to the guest domain is limited, which makes the split driver model a safe approach at first sight. Nevertheless, if the backend driver runs on the privileged domain and not in a separate isolated driver domain, the device driver is part of the Trusted Computing Base (TCB), along with the hypervisor and the hardware. As such, a compromise of the backend driver can lead to the compromise of the entire system and break isolation between guest VMs. Reducing the TCB to its minimum is a common method to improve security. One approach is [38], that breaks the monolithic Gallium 3D graphic driver to move a portion of the code out of the privileged domain. More generally, reducing the TCB is a daunting task given that the TCB of a virtualization platform is already very large [11]. Drivers are well-known to be a major source of operating systems bugs [10]. GPU drivers are also very complex, require several modules and have a large code base. In the case of NVIDIA drivers, code cannot be inspected and verified since it is closed source. Like any complex piece of software, GPU drivers can suffer from vulnerabilities, such as those reported for NVIDIA drivers [1,2,3,4,5].

Direct Device Assignment This technique gives direct access to a physical GPU, with a very limited level of interposition. The PCI passthrough is managed by QEMU and the IOMMU, that become two targets for attacks. The attack surface of the IOMMU is large since it has to handle every calls to the hardware: Memory-Mapped Input/Output (MMIO), Programmed Input/Output (PIO), DMA, interrupts. Although a piece of hardware is generally known as more secure than a piece of software, the IOMMU is prone to attacks [27,43]. Side channels are of less importance because the GPU is not simultaneously shared by two tenants, but information leakage can still occur given that it is physical hardware that is shared across different sessions.

Direct Device Assignment with SR-IOV This setup is recent and not yet deployed by cloud providers, so no study has been conducted to assess its security. Because they are designed for virtualization and for sharing, it is likely that they will provide an isolation mechanism that will prevent *direct* information leakage from a parallel adversary. However, if memory cleaning is not properly implemented, it is the same situation as direct device assignment for a serial adversary. Moreover, performance and resource sharing are antagonistic to side channel resistance. Therefore we can expect that *indirect* information leaks will be possible.

Full emulation and split driver techniques have low maturity and performance, and SR-IOV GPUs are not currently deployed. Therefore, in the rest of this paper we focus on data leaks in virtualization setups when GPUs are used in direct device assignment mode, and in cloud setups. This effectively restricts the adversary model to the *serial adversary.*

4 Experiments Setup

In this section, we detail the experiments that we conducted during our study. We consider the *serial adversary*. We organize our experiments according to two main parameters: the degree of virtualization, and the method used to access the memory.

We pursue experiments with no virtualization, and with direct device assignment GPU virtualization. We use a lab setup for both settings and a real life cloud computing setup using Amazon. In our virtualized lab setup, we test two hypervisors: KVM [21] and Xen [6]. For both of them, we used HVM virtualization, with VT-d enabled. The administrative and guest VMs run GNU/Linux. The cloud computing setup is an Amazon GPU instance that uses Xen HVM virtualization with an NVIDIA Tesla GPU in direct device assignment mode. The VM also runs GNU/Linux.

We pursue experiments accessing the memory with different GPGPU frameworks under different drivers, as we explain in Section 5. We also access the memory with no framework through the PCI configuration space, in a driver agnostic way, as we describe in Section 6. To that extent, we build a generic CUDA *taint* program and two *search* programs, depending on the access method.

1. *Taint* writes identifiable strings in the global memory of the GPU. It makes use of the CUDA primitives `cudaMalloc` to allocate space on the global memory, `cudaMemcpy` to copy data from host to device, and `cudaFree` that frees memory on the device.
2. *Search* scans the global memory, searching for the strings written by *taint*. The program that uses a GPGPU framework operates in the same way as *taint* by allocating memory on the device. However, data is copied from device to host before finally freeing memory. The other program uses the PCI configuration space.

We first execute *taint*, then *search*, with various actions between these two executions. An information leakage occurred if *search* can retrieve data written by *taint*. Table 1 summarizes the experiments and their results.

5 Accessing Memory Through GPGPU Runtime

In this section, we detail our method and results to access the GPU memory with the CUDA and Gdev runtimes, in three environments: Native, virtualized and cloud.

[4] We cannot guarantee that we end up in the same physical machine after releasing a VM in the cloud setup.

[5] The access through PCI configuration space needs root privilege.

Table 1: Overview of the attacks and results. The different actions between *taint* and *search* are: (1) switch user; (2) soft reboot bare machine or VM; (3) reset GPU using `nvidia-smi` utility; (4) kill VM and start another one; (5) hard reboot machine. ✓ indicates a leak, and ✗ no successful leak. N/A means that the attack is not applicable.

		Actions between *taint* and *search*				
Setup	ECC	1	2	3	4	5
GPGPU runtime access						
Native	on	✓	✗	✗	N/A	✗
	off	✓	✓	✓		✗
Virtualized	on	✓	✗	✗	✗	✗
	off	✓	✓	✓	✓	✗
Cloud	on	✓	✗	✗	N/A[4]	N/A
	off	✓	✓	✓		
PCI configuration space access						
Native	on	N/A[5]	✗	✗	N/A	✗
	off		✓	✓		✗
Virtualized	–	N/A[5]	✗	✗	✗	✗
Cloud	–	N/A[5]	✗	✗	N/A[4]	N/A

5.1 Native Environment

We conduct experiments similar to [9,12] with a Quadro Fermi GPU that does not provide ECC for its memory. We validate information leakage on two frameworks: (i) using the runtime API on top of the CUDA runtime and the NVIDIA driver and (ii) using the driver API on top of the Gdev runtime and the Nouveau driver. We observed information leakage when users switch, when there is a soft reboot and when the GPU is reset, *i.e.,* in all cases between *search* and *taint* except for the hard reboot. This indicates that the GPU maintains data in memory as long as it is powered, *i.e.,* anyone can retrieve data during this time. The driver and framework do not impact memory leakage in this setting.

We now consider a Tesla Kepler GPU which provides ECC for its memory. We found that the Tesla GPU has two options that impact the behavior of the memory:

- Persistence mode: Enabling persistence keeps the driver loaded even when no application is accessing the GPU and minimizes the driver load latency.
- ECC mode: When the Error Correction Code option is enabled part of the dedicated memory is used for ECC bits, this reduces the available memory by 12.5%. ECC protects register files, L1/L2 caches, shared memory, and DRAM [29]. It takes effect after the next reboot, or device reset.

Table 2 shows in which cases we could observe an information leakage with a user switch on the Tesla Kepler GPU in a native environment. The only case where we could not observe any information leakage is when ECC is enabled and

Table 2: Information leakage with user switch between the execution of *taint* and *search*, as function of ECC and persistence mode. Tested on a Tesla card in a native environment. ✓ indicates a leak, and ✗ no successful leak.

	ECC enabled	**ECC disabled**
persistence off	✗	✓
persistence on	✓	✓

persistence is disabled. In this mode, the driver loads dynamically each time a GPU application is executed. These experiments suggest that memory cleaning is triggered by loading the driver when ECC is enabled. Furthermore, memory is not zeroed with ECC and persistence disabled; this indicates that memory zeroing in the ECC case is not implemented for security reasons but only to properly support ECC mode.

In the case of a soft reboot of the machine or a reset of the GPU, the driver is unloaded and reloaded independently of the persistence mode. There is no information leakage between *taint* and *search* with ECC enabled in these cases.

5.2 Virtualized Environment

From a guest VM, we observed information leakage when switching user between *taint* and *search*, which is the same behavior as in a native environment. The soft reboot and the GPU reset are also giving different result depending on ECC, showing information leakage when ECC is disabled, and no leakage when ECC is enabled. Consistently with the native environment, there was no information leakage after a hard reboot. Information leakage on these setups threatens the confidentiality between users and applications of the same guest VM.

To investigate the role of the hypervisor, we are interested in knowing whether a guest VM can retrieve data in the GPU memory left by a previous guest VM. For that matter, we create a guest VM running NVIDIA driver on Ubuntu, launch the *taint* program and then destroy the VM. Afterwards, we create another guest VM and launch the *search* program. We could retrieve data on both Xen and KVM, revealing that information has leaked. This result indicates a clear violation of the isolation that the hypervisor must maintain between two guest VMs.

5.3 Cloud Environment

Within the same guest VM, we obtain the same results as in the virtualized environment. Information leakage occurs with ECC disabled when there is a user switch, after a soft reboot of the VM or a reset of the GPU.

In the default configuration of Amazon GPU instances, ECC is enabled and persistence is disabled. In accordance with our previous experiments, it means that the memory is cleaned, and it is supposed to prevent a user from accessing the memory of previous users. However, a user that deactivates ECC to have more memory available (or uses a VM image configured this way) may not be protected. Based on our observations, we imagine an attack where an adversary rents many instances and disables ECC – or provides a custom image that disables ECC to numerous victims. Slaviero et al. [37] showed that it is possible to pollute the Amazon Machine Image market with VM images prepared by an adversary. The adversary then waits for its victim to launch an instance where the ECC has been disabled. When the victim releases the instance, the adversary tries to launch its own instance on the same physical machine. While this may be difficult, Ristenpart et al. [35] showed that it is possible to exploit and influence VM placement in Amazon. The adversary then runs the *search* program to seek data in the GPU memory. We did not implement this attack as we would have needed to rent a large number of instances, without any guarantee to retrieve the same physical machine as a victim's.

We therefore contacted Amazon security team, who mentioned that they were already addressing such concerns in their pre-provisioning workflow, *i.e.,* before allocating a new instance to a user. However, without further details on how GPU memory is cleaned, there is no guarantee that Amazon performs this correctly. In addition to this, in absence of formal industry recommendations, we cannot exclude the existence of data leakage in other GPU cloud providers.

6 Accessing Memory Through PCI Configuration Space

The access method that leverages GPGPU runtime has the disadvantage of only showing a partial view of the GPU memory, *i.e.,* only what can be accessed via the GPU MMU. In this section, we show a method to access the GPU memory through the PCI configuration space, in a driver agnostic way.

6.1 Native Environment

There are two methods to perform I/O operations between the CPU and I/O devices: Memory-Mapped I/O (MMIO) and Port-mapped I/O (PIO). The mapping of the device memory to the MMIO or PIO address space is configured in the Base Address Registers (BAR), in the PCI configuration space. The PCI configuration space is a set of registers that allow the configuration of PCI devices. Reads and writes can be initiated by the legacy x86 I/O address space, and memory-mapped I/O.

For NVIDIA GPUs, the BARs are obtained by a reverse-engineering work of the open-source community. BAR0 contains MMIO registers, documented in the Envytools git [14]. The registers are architecture dependent, but the area we are interested in remains the same for the architectures Tesla, Fermi and Kepler. The mapping at `0x700000-0x7fffff`, called PRAMIN, can be used to access any part of video memory by its physical address. It is used as a 1MB window to physical memory, and its base address can be set using the register `HOST_MEM` at the address `0x1700`. Figure 3 illustrates this access.

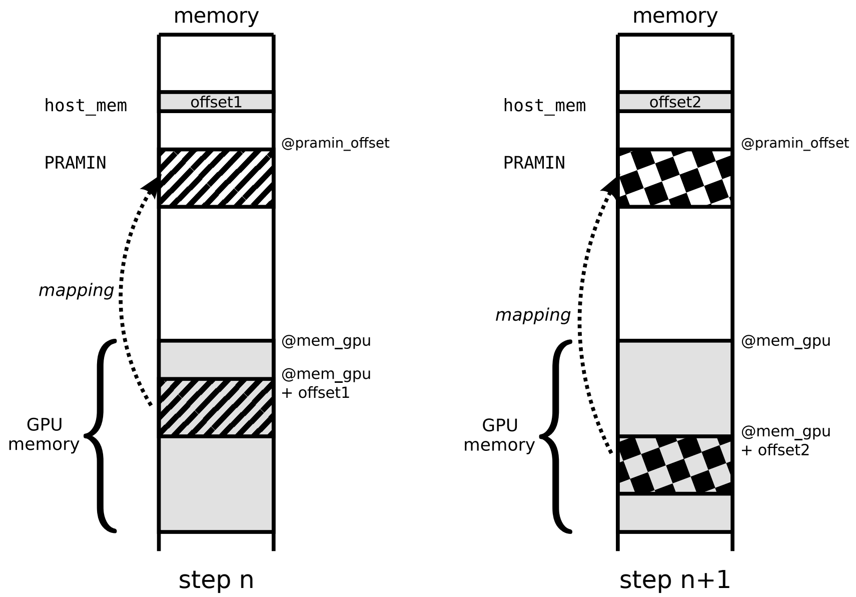

Fig. 3: Accessing GPU memory via PCI configuration space: PRAMIN mapping is used to access 1MB of the GPU physical memory, at address configured in the register `host_mem`. We depict two consecutive steps in Algorithm 1 `while` loop.

Algorithm 1 Accessing memory through PRAMIN

```
pramin_offset ← 0x700000
host_mem ← 0x0
vram[size]
while i < size do
    read(pramin_offset, vram[i], 0x100000)
    host_mem ← host_mem + 0x100000
end while
```

The access to video RAM is done through the following steps. First, `HOST_MEM` is set to `0x0` and we read the 1MB of PRAMIN – that way we are able to read the first 1MB of the GPU's physical memory. We then add 1MB to `HOST_MEM` and re-read PRAMIN. This step is done again until the whole memory has been accessed. Algorithm 1 summarizes these steps. We use read and write functions of the Envytools [14] (`nva_wr32` and `nva_rd8`), that in turn use `libpciaccess` to access the PCI configuration space.

Consistently with the experiments leveraging a GPGPU runtime, we observe information leakage after a soft reboot and a reset of the GPU. There is no information leakage after a hard reboot. Changing user does not apply in this setup since we need to be root to access the PCI configuration space.

Accessing memory through PCI configuration space gives a complete snapshot of the GPU memory and bypasses the GPU MMU. The advantage of such method is that it is capable of bypassing some memory cleanup measures implemented at the applicative level. We discuss this aspect in Section 7.

6.2 Virtualized and Cloud Environment

Xen provides I/O virtualization by means of emulation for its HVM guests with the QEMU device model (QEMU-dm) daemon that runs in Dom0. When a guest is configured with a device in direct device assignment mode, QEMU-dm reads its PCI configuration space register, and then replicates it in a virtual PCI configuration space. QEMU-dm maps MMIO and PIO into the guest memory space, and configures the IOMMU to grant the guest OS access to these memory regions. However, QEMU-dm emulates some configuration space registers like BAR for security reasons, so that an adversary cannot change the memory mapping of the device to another device attached to another VM, or to the hypervisor. Other registers like command register are not emulated.

Our access method leverages BAR registers to access the GPU memory. We tested the method on our Xen setup and obtained garbage (series of `0xffff` values), confirming that the access to the registers are emulated, which prevented us from effectively accessing the memory. The results are the same for Amazon GPU instances. These setups are then showing no information leakage. To circumvent the protection of BAR registers, an adversary may try to attack the virtualization mechanisms themselves.

7 Countermeasures

We divide the possible countermeasures in three categories: changes in existing runtimes, steps that can be taken by cloud providers, and those that can already be initiated by a user using only calls to existing APIs.

Changes to Existing Runtimes Di Pietro et al. [12] suggest an approach to be implemented in runtimes. The solution is to zero-fill buffers at allocation time, as it is done when an operating system allocates a new physical page of memory to a process. This solution targets an adversary that uses GPGPU runtime to launch her attack, however, it does not protect from an adversary that accesses memory through PCI configuration space, since she will not allocate memory. In this case, it would be better to clear memory at deallocation time. In both cases, zero-filling buffers entails performance issues as the memory bandwidth is generally a bottleneck for GPGPU applications. Di Pietro et al. assess the impact of the `cudaMemset` function that is used for zeroing buffers. The overhead turns out to be linearly proportional to the buffer size.

Cloud Providers Cloud providers can already take measures to protect their customers. The necessary steps before handing an instance to a customer include cleanup of the GPU memory. This is the approach that appears to be taken by Amazon, which seems to implement proper memory cleaning and does not rely solely on a side effect of having ECC enabled by default.

Defensive Programming In the absence of the two types of countermeasures above, a security-conscious programmer that writes his own kernels and can accept a performance penalty can clear the buffer before freeing memory with a function such as `cudaMemset`. If the end-user can not modify the program, he should erase the GPU memory when finishing an execution on a GPU. This countermeasure seems trivial, nevertheless its practical implementation can be difficult due to the complicated memory hierarchy present in GPUs (e.g., access mechanisms depend on the type of memory). A standalone CUDA program that cleans the memory would allocate the maximum amount of memory, and then overwrite it (e.g., with zeros). However, this solution relies on the CUDA memory manager, which does not guarantee the allocation of the whole memory. Portions of memory risk not to be properly erased because of fragmentation issues. We built an experiment to illustrate this: We run a CUDA program for some time, then we stop it to run the CUDA program that cleans the memory. We finally dump the memory via PRAMIN to access the whole memory. We clearly recovered a portion of the memory that was not cleaned by the CUDA program, demonstrating clear limitations of this countermeasure.

A practical solution for NVIDIA Tesla GPUs that benefit from ECC memory is to enable ECC and reload the driver, or to reset the GPU when ECC is enabled. As we saw in our experiments Section 5.1, these sequences of actions clear the memory.

8 Conclusions

We evaluated the confidentiality issues that are posed by the recent advent of GPU virtualization. Our experiments in native and virtualized environments showed that the driver, operating system, hypervisor and the GPU card itself do not implement any security related memory cleanup measure. As a result, we observed information leakage from one user to another, and in particular from one VM to another in a virtualized environment. Amazon seems to implement proper GPU memory cleaning at the provisioning of an instance; we could thus not confirm any information leakage from one Amazon instance to another. However, because of the general lack of GPU memory zeroing, we cannot generally exclude the existence of data leakage in cloud computing environments.

The rise of GPGPU increases the attack surface and urges programmers and industry to handle GPU memory with the same care as main memory. For this matter, industry should include GPU memory cleaning in its best practices. We provided a set of recommendations for proper memory cleanup at the various layers involved in GPU virtualization (application, driver, hypervisor).

In the future, GPU virtualization will move from sequential sharing of a GPU card to simultaneous sharing between several tenants. Proper memory isolation will become even more challenging in this context, and we plan to study this aspect in future work.

Acknowledgments

We wish to thank NVIDIA for the donation of a Tesla K20 card. We would also like to thank the Nouveau development team, and especially Martin Peres, for sharing their knowledge and their massive effort of reverse-engineering on NVIDIA GPUs.

References

1. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-0946`, 2012.
2. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-4225`, 2012.
3. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-0109`, 2013.
4. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-0110`, 2013.
5. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-0131`, 2013.
6. P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.
7. M. Becchi, K. Sajjapongse, I. Graves, A. Procter, V. Ravi, and S. Chakradhar. A virtual memory based runtime to support multi-tenancy in clusters with GPUs. In *HPDC'12*, 2012.
8. A. Bernemann, R. Schreyer, and K. Spanderen. Pricing structured equity products on gpus. In *Workshop on High Performance Computational Finance (WHPCF'10)*, 2010.
9. S. Breß, S. Kiltz, and M. Schäler. Forensics on GPU Coprocessing in Databases – Research Challenges, First Experiments, and Countermeasures. In *Workshop on Databases in Biometrics, Forensics and Security Applications*, 2013.
10. A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *SOSP'01*, 2001.
11. P. Colp, M. Nanavati, J. Zhu, W. Aiello, G. Coker, T. Deegan, P. Loscocco, and A. Warfield. Breaking Up is Hard to Do: Security and Functionality in a Commodity Hypervisor. In *SOSP'11*, 2011.
12. R. Di Pietro, F. Lombardi, and A. Villani. CUDA Leaks: Information Leakage in GPU Architectures. *arXiv:1305.7383v1*, 2013.
13. M. Dowty and J. Sugerman. GPU virtualization on VMware's hosted I/O architecture. *ACM SIGOPS Operating Systems Review*, 43(3):73–82, 2009.
14. Envytools. `https://github.com/envytools/envytools`.
15. G. Giunta, R. Montella, G. Agrillo, and G. Coviello. A GPGPU Transparent Virtualization Component for High Performance Computing Clouds. In *Euro-Par'10*, 2010.
16. gKrypt Engine. `http://gkrypt.com/`.
17. V. Gupta, A. Gavrilovska, K. Schwan, H. Kharche, N. Tolia, V. Talwar, and P. Ranganathan. GViM: GPU-accelerated Virtual Machines. In *HPCVirt'09*, 2009.
18. D. Harnik, B. Pinkas, and A. Shulman-peleg. Side channels in cloud services, the case of deduplication in cloud storage. *IEEE Security & Privacy*, 8(6):40 –47, 2010.
19. S. Kato, M. McThrow, C. Maltzahn, and S. Brandt. Gdev: First-Class GPU Resource Management in the Operating System. In *USENIX ATC'12*, 2012.
20. M. Kerrisk. Xdc2012: Graphics stack security. `https://lwn.net/Articles/517375/`, 2012.

21. A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm : the Linux Virtual Machine Monitor. In *Proceedings of the Linux Symposium*, pages 225–230, 2007.
22. C. Kolb and M. Pharr. *GPU Gems 2*, chapter Options Pricing on the GPU. 2005.
23. C. Kolivas. cgminer. `https://github.com/ckolivas/cgminer`.
24. E. Ladakis, L. Koromilas, G. Vasiliadis, M. Polychronakis, and S. Ioannidis. You Can Type, but You Can't Hide: A Stealthy GPU-based Keylogger. In *EuroSec'13*, 2013.
25. E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. Nvidia Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, 2008.
26. F. Lombardi and R. Di Pietro. CUDACS: Securing the Cloud with CUDA-Enabled Secure Virtualization. In *ICICS'10*, 2010.
27. F. Lone Sang, E. Lacombe, V. Nicomette, and Y. Deswarte. Exploiting an I/OMMU vulnerability. In *MALWARE'10*, 2010.
28. Nouveau. `http://nouveau.freedesktop.org`.
29. NVIDIA. TESLA M2050 / M2070 GPU computing module, 2010.
30. NVIDIA. CUDA C Programming Guide, 2012.
31. NVIDIA. NVIDIA GRID, GPU Acceleration for Virtualization, GPU Technology Conference. `http://on-demand.gputechconf.com/gtc/2013/presentations/S3501-NVIDIA-GRID-Virtualization.pdf`, 2013.
32. R. Owens and W. Wang. Non-interactive OS fingerprinting through memory deduplication technique in virtual machines. In *IPCCC'11*, 2011.
33. Pathscale. `https://github.com/pathscale/pscnv`.
34. V. T. Ravi, M. Becchi, G. Agrawal, and S. Chakradhar. Supporting GPU Sharing in Cloud Environments with a Transparent Runtime Consolidation Framework. In *HPDC'11*, 2011.
35. T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *CCS'09*, 2009.
36. L. Shi, H. Chen, and J. Sun. vCUDA: GPU accelerated high performance computing in virtual machines. In *IPDPS'09*, 2009.
37. M. Slaviero, H. Meer, and N. Arvanitis. Clobbering the Cloud, part 4 of 5, Blackhat. `http://www.sensepost.com/blog/3797.html`, 2009.
38. C. Smowton. Secure 3D graphics for virtual machines. In *EuroSec'09*, 2009.
39. K. Suzaki, K. Iijima, T. Yagi, and C. Artho. Memory Deduplication as a Threat to the Guest OS. In *European Workshop on System Security*, 2011.
40. X. Tian and K. Benkrid. High-performance quasi-monte carlo financial simulation: FPGA vs. GPP vs. GPU. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 3(4):26, 2010.
41. G. Vasiliadis, M. Polychronakis, and S. Ioannidis. GPU-assisted malware. In *International Conference on Malicious and Unwanted Software*, 2010.
42. M. S. Vinaya, N. Vydyanathan, and M. Gajjar. An evaluation of CUDA-enabled virtualization solutions. In *PDGC'12*, 2012.
43. R. Wojtczuk and J. Rutkowska. Following the White Rabbit: Software attacks against Intel VT-d technology. *invisiblethingslab.com*, 2011.
44. Z. Wu, Z. Xu, and H. Wang. Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud. In *USENIX Security*, 2012.
45. T. Yamanouchi. *GPU Gems 3*, chapter AES Encryption and Decryption on the GPU. 2007.
46. Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-VM side channels and their use to extract private keys. In *CCS'12*, 2012.