Understanding The Security of Discrete GPUs

Zhiting Zhu
The University of Texas at Austin
zhitingz@cs.utexas.edu

Sangman Kim
The University of Texas at Austin
sangmank@cs.utexas.edu

Yuri Rozhanski
Technion-Israel Institute of Technology
syuriro@t2.technion.ac.il

Yige Hu
The University of Texas at Austin
yige@cs.utexas.edu

Emmett Witchel
The University of Texas at Austin
witchel@cs.utexas.edu

Mark Silberstein
Technion-Israel Institute of Technology
mark@ee.technion.ac.il

CCS Concepts •Security and privacy → Operating systems security;

ACM Reference format:
Zhiting Zhu, Sangman Kim, Yuri Rozhanski, Yige Hu, Emmett Witchel, and Mark Silberstein. 2016. Understanding The Security of Discrete GPUs. In *Proceedings of GPGPU-10, Austin, TX, USA, February 04-05 2017,* 11 pages.
DOI: http://dx.doi.org/10.1145/3038228.3038233

Abstract

GPUs have become an integral part of modern systems, but their implications for system security are not yet clear. This paper demonstrates both that discrete GPUs cannot be used as secure co-processors and that GPUs provide a stealthy platform for malware. First, we examine a recent proposal to use discrete GPUs as secure co-processors and show that the security guarantees of the proposed system do not hold on the GPUs we investigate. Second, we demonstrate that (under certain circumstances) it is possible to bypass IOMMU protections and create stealthy, long-lived GPU-based malware. We demonstrate a novel attack that compromises the in-kernel GPU driver and one that compromises GPU microcode to gain full access to CPU physical memory. In general, we find that the highly sophisticated, but poorly documented GPU hardware architecture, hidden behind obscure close-source device drivers and vendor-specific APIs, not only make GPUs a poor choice for applications requiring strong security, but also make GPUs into a security threat.

1 Introduction

GPUs have enjoyed increasing popularity over the past decade, both as hardware accelerators for graphics applications and as highly parallel general-purpose processors. With general purpose computing on GPUs (GPGPUs) diffusing into the mainstream, researchers are looking at their security implications. In this paper we analyze two related questions: can GPUs be used to enhance the security of a computing platform? and can GPUs be used to subvert the security of a computing platform?

Understanding the security of GPUs requires understanding the interplay among the GPU hardware, its software stack, and the busses and chipsets that coordinate a platform's transfer of data. The interplay of these features is complicated by GPU hardware that contains quirky features absent from CPUs, such as auxiliary embedded microprocessors, and by the GPU's deep software stack whose boundaries and interactions with GPU hardware are deliberately blurred by the vendor. Unfortunately, the complexity of this interplay can hide vulnerabilities that attackers can use to subvert a GPU's expected behavior and break critical security properties, as we show in this paper.

For discrete GPUs, their independent memory system and computational resources are physically partitioned from the main CPU which makes it plausible that a GPU could function as a secure processor; it might be possible to protect computation on a discrete GPU from code executing on the CPU. While plausible in theory, we systematically analyze the shortcomings of one specific proposal to use GPU hardware registers as secure storage (called PixelVault [44]). We show that PixelVault's security depends on assumptions about GPU hardware features that do not hold, and in practice fully depend on the vulnerable GPU *software* interface that GPU vendors expose.

The flaws we find in PixelVault's GPU security model stem from the lack of a clear software/hardware boundary and shifting responsibilities of hardware and software across GPU generations. For example a non-bypassable hardware feature in one version of a GPU can migrate to a bypassable software feature in another version. The problem with such a fluctuating software/hardware boundary is that it becomes hard, if not impossible, to reason about the actual security guarantees of a GPU system.

We also systematically analyze risks that originate with NVIDIA GPUs, where the GPU serves as a host for *stealthy, long-lived* malicious code. It is difficult to detect the execution of GPU-hosted malware and in certain cases, it is even difficult to detect its presence. We demonstrate attack code running on the NVIDIA GPU that reads secrets from CPU memory and corrupts the memory state of CPU computations by leveraging GPU Direct Memory Access (DMA) capabilities.

We demonstrate two novel attacks: one against the proprietary in-kernel closed-source GPU driver, the other against the GPU microcode running on an auxiliary microprocessor resident on the GPU card. For the driver attack, we binary patch the proprietary NVIDIA GPU driver while it is loaded and being used by the OS kernel, and force it to map sensitive CPU memory into the address space of an unprivileged GPU program. Our second attack leverages auxiliary microprocessors [27, 32] which GPUs use for various functions like power management and video display management. These microprocessors are not exposed as part of the standard GPU programming model (e.g., CUDA or OpenCL). We implement the attack microcode running on such an auxiliary microprocessor that combines the functionality of the original microcode with malicious

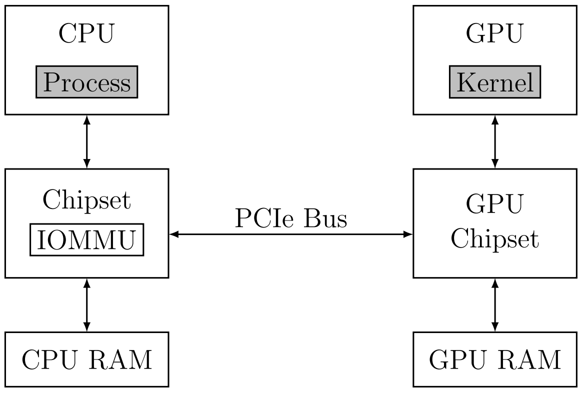

Figure 1. CPU-GPU architecture overview. IOMMU use is optional and its behavior is configured by the operating system.

code that can read and write arbitrary CPU memory. To the best of our knowledge it is the first such attack on NVIDIA GPUs.

Our attacks rely on unlimited DMA access from GPU to CPU physical memory. The common way to foil such DMA attacks has been to restrict peripheral access to system memory by using the IO memory management unit (IOMMU), intended to protect CPU memory against malicious or malfunctioning peripherals. We, however, show novel techniques to subvert IOMMU security and bypass its protection in the strict operation mode that has previously been considered the most secure (§4).

Similar DMA attacks on other peripherals are known [10, 11, 34, 40] However, our GPU DMA attacks are particularly dangerous because they are relatively easy to program: the ability of modern GPUs to execute general purpose code lowers the bar for implementing sophisticated malware on GPU.

Our attacks assume an adversary who transiently gains the ability to load a kernel module. The primary danger of the GPU-based attacks is their *stealthiness*. They would evade most known tools and research proposals that provide software system integrity.

Our work focuses on discrete GPUs from NVIDIA which come mounted on an expansion card which is plugged into a computer's IO bus. The only current proposal to use a GPU as a secure processor is for a discrete GPU. While integrated GPUs also use IOMMUs for safety, we leave for future work the dangers and mitigations for integrated GPU security.

We begin by summarizing the portions of the GPU architecture that are most relevant for security (§2), and then describe how we can defeat the security of PixelVault (§3). Then we discuss how to bypass IOMMU protections (§4) and attack CPU memory using a compromised GPU driver (§5) and compromised microcode running on an embedded GPU microprocessor (§6). We summarize related work (§7) and conclude.

2 Architecture

This section summarizes relevant aspects of GPU architecture, paying close attention to the memory subsystem and CPU-GPU communications. We describe *discrete* GPUs that connect to the host via a peripheral component interconnect express (PCIe) bus, because this paper focuses on discrete GPUs, and in particular on NVIDIA GPUs. A high-level view of a CPU-GPU system model used throughout this paper is shown in Figure 1. Process and kernel are shaded, because they are software abstractions.

2.1 GPU execution model

GPUs are slave processors controlled entirely by a CPU executing a GPU driver running in privileged mode. Any CPU process can initiate the execution of a GPU program by making API calls to the kernel-resident GPU driver. The initiating CPU process is called the *GPU-controlling* process. The unprivileged CPU process invokes a GPU *kernel*, which is specially written and compiled for execution on a GPU.

There are public APIs to allow the CPU-controlling process to manipulate the address space of any GPU kernel that it launches. Both NVIDIA CUDA and OpenCL contain APIs that allow management of the GPU kernel's memory, including transfer of data from/to the CPU, and mapping CPU memory into a kernel's address space, as we discuss next. Once the GPU-controlling process terminates, all the GPU resources associated with it are released. In particular, the driver reclaims the GPU memory and terminates active GPU kernels associated with the process.

2.2 Memory Hierarchy

NVIDIA GPUs contain several streaming multiprocessors (SMs), which concurrently run thousands of sequential threads. Each thread may access its private registers, local on-die scratchpad memory, and global GPU memory shared among all SMs. Global memory is cached with two levels of hardware cache. The L1 data cache is local for each SM, while the L2 is shared across all SMs.

Instruction cache. NVIDIA GPUs have multiple levels of instruction cache, though the exact architecture has not been officially disclosed. Wong et al. [48] suggest that there are three levels. The instruction caches in GPUs are used exclusively for instructions. Further, the instruction cache is not kept coherent with the GPU global memory where the instructions are stored. The GPU driver flushes the instruction cache when a new GPU program starts. However, if the CPU writes to GPU instructions in memory while the GPU is running, the (stale) instructions are not invalidated from the cache. NVIDIA does not provide any public API for flushing the instruction cache. Therefore, overwriting a GPU kernel's program code in GPU memory while a GPU is running may not change the actual running program.

2.3 Accessing CPU memory from the GPU

Modern GPUs provide limited access to CPU memory from programs running on a GPU. In particular, a standard API (`cudaHostRegister` for CUDA [33] and `clEnqueueMapBuffer` for OpenCL [23]) allows the CPU to map a CPU memory region into a GPU kernel's address space. Once mapped, the GPU may directly access the mapped CPU memory without CPU involvement via direct memory access (DMA). Similarly, GPUs may be configured to access memory-mapped input/output (MMIO) regions of other *peer* peripheral devices connected to the PCIe bus. For example, the NVIDIA GPUDirectRDMA API enables peer-to-peer access to Infiniband network cards, allowing GPU programs to communicate over the network without CPU mediation [31]. The GPU internal page table is generally accessible only through the GPU driver and is not visible to the CPU OS.

In contrast to CPU programs, in which memory protection is enforced for every load, store and instruction execution at runtime by hardware, GPU accesses to CPU memory do not pass through the CPU's memory management unit (MMU) and therefore the CPU performs *no* runtime checks. Rather, the GPU driver validates access rights at the time of *mapping* in *software*. Additional hardware protection can be provided by the IOMMU which we describe next.

2.4 IOMMU

When a device performs direct memory access (DMA) to read or write CPU physical memory, it uses device addresses. When the IOMMU is working, it maps device addresses to CPU physical addresses (just as the CPU's MMU maps virtual to physical address). The IOTLB caches entries from the IO page table, just as the CPU's TLB caches entries from the process' page table. IO page table entries contain protection information, and the IOMMU will check each access to system memory from a peripheral device, to make sure it has sufficient permissions.

The IOTLB is not kept coherent with the IO page table by hardware, similarly to TLBs in most common CPUs. Software must explicitly manage the IOTLB, flushing the cached mappings when they are removed from the IO page table. We exploit this software-managed IOTLB coherence mechanism to circumvent IOMMU protection and enable unauthorized access to system memory from the GPU, as we discuss in Section 4.

2.5 Microprocessors and MMIO registers in GPUs

GPUs expose a set of memory mapped input output (MMIO) registers used by the driver for GPU management [2, 27]. In addition, they contain several special-purpose microprocessors used to manage internal hardware resources. A GPU driver updates GPU microprocessor code every time a GPU is initialized. The documentation about the actual purpose of microprocessors and MMIO registers used in NVIDIA GPUs is fairly scarce; it usually comes from unofficial sources, such as open-source driver developers who partially reverse-engineered the official driver.

We found that the GPU MMIO registers can invalidate the GPU instruction caches. Flushing the instruction caches is key to dynamically updating the code of a running kernel, which breaks the security guarantees of PixelVault (§ 3.2). Our microcode attack (§ 6) leverages an important capability of NVIDIA microprocessors that allows unrestricted access to GPU and CPU memory [17].

3 Attacking PixelVault

In this section we analyze the GPU model and security guarantees PixelVault uses to claim a GPU as a secure co-processor. We then present several attacks that clearly violate PixelVault's assumptions, and therefore its security properties. We conclude that systems developed using PixelVault's approach are insecure.

Experimental platform. The attacks described in Section 3.4 and Section 3.3 are performed on NVIDIA Tesla C2050/C2075 GPU (Fermi) and an NVIDIA GK110GL Tesla K20c (Kepler), using NVIDIA driver versions 319.37 and 331.38 respectively, and CUDA version 5.5. The attack in Section 3.2 is performed on an NVIDIA Tesla C2050/C2075 (Fermi) with the open source nouveau [29] and gdev [21] [22]) drivers.

3.1 PixelVault summary and guarantees

PixelVault proposes a GPU-based design of a security co-processor for RSA and AES encryption which is resilient to even a strong adversary with full control of CPU and/or GPU software. PixelVault stores the secret keys encrypted in GPU memory, and the master key in GPU registers. It implements a software infrastructure that strives to prevent any adversarial access to these registers from CPU or GPU.

PixelVault threat model. PixelVault assumes that the system boots from a trusted configuration, and it can set up its execution environment on the GPU. Once PixelVault is established, the attacker may have *full control* over the platform. Specifically, the attacker can execute code at any privilege level and it has access to all platform hardware.

To achieve this goal PixelVault leverages several characteristics of the NVIDIA GPU architecture and execution model. While some of these characteristics are well known and have been officially confirmed, some were only *assumed* to be correct and others were partially validated experimentally.

Below we list only those assumptions that we later experimentally refute. Even if only one of these assumptions is not satisfied, PixelVault is no longer able to guarantee the secrecy of the master encryption key under its threat model.

1. It is impossible to replace the code of a running GPU kernel if the code is fully resident in the instruction cache. This feature is critical to ensuring that an adversary cannot replace the PixelVault GPU code without stopping the kernel, and therefore without losing the master key stored in GPU registers. We show that NVIDIA GPUs have unpublished MMIO registers that flush the instruction cache, allowing replacement of code from running kernels that are as small as 32B (4 instructions).
2. The contents of GPU registers cannot be retrieved after kernel termination. This feature is essential to PixelVault's ability to prevent a strong adversary from retrieving a master key by stopping a running PixelVault kernel. We show that under certain conditions it is possible to retrieve the contents of registers after kernel termination, and the PixelVault design satisfies these conditions.
3. A running GPU kernel cannot be stopped and debugged if it is not compiled with explicit debug support. This feature is necessary to ensure that an adversary cannot retrieve register contents by attaching a GPU debugger to the running PixelVault kernel. We show that newer versions of the NVIDIA CUDA runtime provide support for attaching a debugger to any running kernel and it is unclear how to disable this capability. This attack requires root privileges to attach to any running process, yet this is permitted under PixelVault threat model.

In the remainder of this section, we explain in more details how we have invalidated the listed assumptions, thereby debunking PixelVault's security.

3.2 Replacing PixelVault as it runs

To run a kernel, a GPU needs the binary code to be resident in GPU memory. The binary is transferred from CPU memory to GPU memory by the driver prior to kernel invocation. GPUs do not support code modification while the kernel is executing. Therefore, PixelVault assumes that GPU hardware makes it impossible for an attacker to alter the execution of a running PixelVault kernel by replacing the original PixelVault binary in GPU memory, as long as the binary is entirely resident in the instruction cache. PixelVault explicitly validates that its kernel is small enough to fit in the instruction cache. Assuming the kernel is simple, PixelVault also explores all possible execution paths to make the kernel fully resident in the cache soon after starting execution.

We find that the lack of software interfaces for dynamic code update does not imply the lack of hardware support. Using the open source Envytools [13] reverse engineering toolkit, we can invalidate the instruction caches and replace the instructions of the *running* kernel with the attacker's modified instructions from GPU memory.
Technical details. We perform an experiment to show that we can dynamically update GPU kernel code using a matrix addition kernel and an updater process. The updater process locates the GPU kernel code in GPU physical memory by searching for the kernel's instructions. In the case of PixelVault, the binary is not secret, so it can be detected by an attacker. Once invoked, PixelVault kernel runs indefinitely, giving the updater enough time to identify and update its code. If the code is erased from memory, the attacker can speculate on its location, methodically working through the address space.

The updater replaces the addition instructions in our test kernel with subtraction instructions. When the effective size of the code in the GPU kernel loop is larger than 32 KB, overwriting the instructions in memory causes the behavior of the kernel to change. Such a large kernel (presumably larger than the size of the last level instruction cache) experiences instruction cache misses at runtime, yielding a result that is not a simple matrix addition.

However overwriting the kernel's program has no effect if the size of the kernel's main loop is smaller than 32KB. In these cases, only if we flush the instruction cache via GPU MMIO registers do we see the expected change in the kernel output.

Our prototype requires 3.1 seconds to scan and identify kernel code in GPU memory. Therefore, we can only effectively flush the cache for long running kernels. The PixelVault kernel is intended to run continuously as it provides a runtime encryption service, therefore it is vulnerable to this cache flush attack.
MMIO registers for instruction cache flush. To invalidate the L1 instruction cache with the updated code memory, it is necessary to flush all the cache levels. The addresses in parentheses are the offsets within the MMIO region, which is referred by the first set of the PCI base address registers (BAR0) of NVIDIA GPUs. The register that flushes the per-GPC caches is `PGRAPH.GPC_BROADCAST.CCACHE.CACHE_CTRL` (0x419000), especially the first bit of the 32-bit register. For the per-SM flush, we used the first and ninth bits of `PGRAPH.GPC_BROADCAST.TPC_ALL.MP.CCACHE_CTRL` (0x419ea4). To the best of our knowledge, the use of the ninth bit of `CCACHE_CTRL` for flushing per-SM cache is not reported or documented.

3.3 Capturing PixelVault secrets after termination

PixelVault relies on GPU registers being initialized to zero when a new kernel is loaded onto a GPU and begins execution. Initial zero values for registers is necessary to prevent an adversary from terminating the PixelVault kernel and running a new kernel that looks for PixelVault secrets in its initial register values. Because initial zero values is not a feature officially documented by NVIDIA, [1] the PixelVault developers experimentally validate that the register contents cannot be retrieved by another kernel invoked after PixelVault terminates. Yet, there is no guarantee that GPU hardware clears registers after termination of a GPU kernel.

We find that a `cuda-gdb` debugger can retrieve register values even after kernel termination. However it requires that other GPU tasks are concurrently active with the execution of the victim kernel. Specifically, NVIDIA CUDA enables multiple GPU operations such as CPU-GPU memory transfers or GPU invocations to be invoked concurrently by the same CPU process. Each operation is invoked in its own *CUDA stream*, and a GPU handles the operations in different streams concurrently. We found that if GPU kernel B is invoked in parallel with running kernel A, A's register state can be retrieved using the debugger API even after A terminates, as long as B is still running.

The PixelVault implementation employs two CUDA streams, one for kernel execution and another for data transfers between a CPU and a GPU. An attacker may take advantage of the data transfer stream to invoke a long-running kernel, terminate PixelVault, and retrieve its secrets.
Technical details. We modified the `cuda-gdb` source code to read the registers of a terminated kernel. The modifications were necessary because by default `cuda-gdb` will refuse to read the registers of a terminated kernel. If we launch two kernels on different CUDA streams, `cuda-gdb` can read the register values from the terminated kernel so long as the other kernel is running. As soon as both kernels terminate, we cannot access either of their registers.

3.4 Stopping PixelVault with a debugger

The PixelVault version discussed in their paper runs NVIDIA CUDA version 4.2. This version of CUDA provided no support for attaching and setting a breakpoint in a *running* GPU kernel, unless that kernel was explicitly compiled with debug information. This property of the runtime environment disguised itself as a hardware feature. Therefore, PixelVault relied on it to ensure that an adversary cannot attach a debugger to retrieve the values of GPU registers which store the secret keys, without stopping the running PixelVault GPU kernel and consequently without erasing the contents of those registers.

However, with the GPU system software evolving so rapidly, many desirable features like attaching a debugger to a running kernel are added in every new release. In particular, this feature was added in the `cuda-gdb` GPU debugger starting from CUDA version 5.0 [18]. By using the CUDA debug API it is possible to stop a kernel, and inspect all GPU registers of the executing kernel from the CPU. This ability invalidates the privacy guarantees for PixelVault's master encryption key.

We found no simple way of preventing software from being able to attach to a running kernel. When attaching, `cuda-gdb` needs access to certain predefined memory locations stored as symbols in `libcuda.so` [30], which is the main library providing CUDA driver API support to GPU applications. In an older version of the CUDA driver (we tested version 319.37), the attach information resides in the `symtab` section and it can be safely stripped. However, more recent versions of the library (we tested version 331.38), no longer place the attach information in `symtab`, they place it in the `dynsym` section. The `dynsym` section cannot be stripped from the binary because it holds important data necessary for dynamic linking. If we remove the `dynsym` section from the binary, the dynamic linker can no longer load `libcuda.so`.

It is possible to zero out the entries in the `dynsym` section used by `cuda-gdb`, which causes `cuda-gdb` to crash the controlling CPU process when attaching to the running GPU kernel. PixelVault could make its own copy of `libcuda.so` (so that other users can

[1] http://docs.nvidia.com/cuda/parallel-thread-execution/index.html#state-spaces-types-and-variables

continue to debug their kernels) and just zero out or corrupt the attach information in `dynsym`. Ultimately however, the ability to stop the running kernel is a hardware feature that PixelVault cannot disable. Because the PixelVault threat model assumes the attacker controls the host, the attacker can still attach to a running kernel and examine its register state even if the default support for how `cuda-gdb` attaches is removed from a version of `libcuda.so`.

Technical details. We use `cuda-gdb` to attach to a running GPU kernel, and retrieve all GPU registers via the CUDA debugger call `CUDBGResult (*CUDBGAPI_st::readRegister)` even if the CUDA application is compiled without debug information (i.e., without the `-G` flag for the NVCC compiler).

A simple experiment verifies this attack. A CUDA application launches a kernel with one thread that spins in an infinite loop and continuously changes a value stored in a register. We attach `cuda-gdb` to the GPU-controlling CPU process and then attach to the GPU kernel that the process spawned. All register values from the running kernel can be extracted whether or not the GPU kernel was built with debug information.

Using this technique, we can also attack a more realistic kernel. We implement the AES encryption algorithm in a GPU program, emulating part of PixelVault's operation. The AES key is stored in GPU registers. While the kernel is running, we attach to it using `cuda-gdb`, read the GPU registers, and expose the secret key.

3.5 Discussion

Discrete GPUs appear to have potential as secure coprocessors because they have physically distinct and complete processing resources: processor, caches, RAM, and access to I/O. They also have micro-architectural (though seemingly robust) guarantees about non-preemption and an incoherent instruction cache. The PixelVault system is an intelligent attempt at trying to build a secure system from these components.

However, our investigation yields the clear conclusion that GPUs are not appropriate as secure coprocessors and cannot contribute to the trusted computing base (TCB) of the system. GPUs are complex devices that rely on sophisticated proprietary hardware and software which is poorly (often purposefully so) publicly documented – the opposite of a firm basis for security. GPU manufacturers are not interested in exposing their architecture internals, and they can easily change the architecture in ways that invalidate the security of systems based on a GPU, e.g., by adding preemption.

We have found a variety of documented and undocumented ways of violating the security of PixelVault. We learned of the existence of many MMIO registers from the Envytools GPU reverse engineering project [13]. However, some registers that allowed us to invalidate the GPU instruction cache are not documented as flushing the cache—yet another example of an obscure GPU architectural subtlety which undermines its use as a secure coprocessor.

4 Threat model and IOMMU

We explore how GPUs might host stealthy malware. Our malware attacks compromise the privacy and integrity of system memory by reading and writing it from the GPU. First we specify our threat model.

4.1 Threat model

An attacker may load and unload kernel modules. In Linux, this can be done by briefly gaining the CAP_SYS_MODULE capability (which is a user credential stored in the process control block) using kernel exploits (e.g., [4, 6, 7]), by bypassing capability checks (e.g., [5, 8, 9]), or by exploiting kernel module loader weaknesses [12]. After loading a module, the attacker also has access to the GPU control interface i.e., MMIO register regions, and as we explain in our microcode attack (§6), it can use these registers to load malicious microcode onto an embedded auxiliary GPU processor. Loading the microcode is done by reloading the GPU driver module into the OS kernel. After the malware is installed, the attacker loses the module loading capability and is allowed *only unprivileged access*.

If an attacker can load a module, why should he or she bother with the attacks we describe? The primary reason is *stealthiness*: all of our attacks originate with the GPU reading and writing CPU memory (e.g., sensitive operating system data structures), making them hard to detect. Detecting root-level compromise is the subject of much published work (e.g., [20], [36], [35], [3]), open source tools (e.g., chkrootkit) and commercial tools (e.g., Malwarebytes AntiRootkit, McAfee Rootkit Remover). To our knowledge, there are no tools and precious few research studies to detect GPU-based malware. Our attacks require no changes to the page table of any unprivileged process, and they bypass the CPU's MMU and memory protection settings. The GPU page table that stores the mappings into system memory are not visible from the CPU (at least not via the public API). Therefore, once mapped into the GPU, a malicious unprivileged GPU kernel may keep accessing any CPU memory without raising suspicion.

Modern systems, however, usually contain an input/output memory management unit (IOMMU) which monitors devices' Direct Memory Accesses (DMAs) to system memory in order to protect it from unauthorized accesses. The IOMMU restricts the devices to access only the CPU memory pages specified in its I/O page table. Unlike the hidden GPU page tables, the I/O page table can be monitored by security tools (though we are not aware of any that do), undermining the stealthiness of the attack.

The malware, therefore, must circumvent IOMMU protection to evade detection, and the next section details the techniques to accomplish that.

4.2 IOMMU

We exploit the subtleties of IOTLB management in the Linux kernel and our prototype is based on the Intel IOMMU. We first provide a brief overview of the IOMMU management policies.

IO device drivers strive to make all memory mappings as short-lived as possible to increase security at the expense of higher management overheads [26]. The OS, therefore, offers several IOMMU configurations that influence the IOTLB management policy and enable different tradeoffs between management cost and security. The configurations and their respective management policies are summarized in Table 1.

IOMMU disabled. Though it is detrimental to security, many systems, especially those that include discrete GPUs, disable their IOMMU by default. IOMMU support for Intel chipsets must be configured through the BIOS as part of setup for device virtualization technology (VT-d), and it also must be enabled in the Linux kernel. Some server manufacturers ship their products with VT-d disabled in the BIOS by default [14].

Several major Linux distributions (e.g., Ubuntu 15.04, CentOS 7, RHEL 7, OpenSUSE 13.2) ship with Intel's IOMMU disabled in the kernel. The primary reasons for disabling the IOMMU is reduced I/O performance [49] and the IOMMU's incompatibility with certain devices and features. For example, the peer-to-peer DMA

Mode	**IOTLB Flush**	
	Strategy	**Timing**
Disabled	None	NA
Pass through	None	NA
Deferred	Flush entire IOTLB.	When deferred list is full or t ms after the first entry [2], whichever comes first.
Strict	Flush individual entry in given domain.	Immediately after unmapping entry from IO page table.

Table 1. Different IOMMU modes, their properties and security characteristics.

capability of NVIDIA GPUs, essential for high I/O performance in multi-GPU systems, requires the IOMMU to be disabled [31].

While we state the obvious, when the IOMMU is disabled, it does not protect CPU memory from malicious accesses performed from the GPU.

IOMMU pass through mode. In pass through mode, device addresses are used directly as CPU physical addresses. In this mode the hardware IOMMU is turned off, so there is no permissions checking for DMA requests. Devices enter pass through mode if it is enabled by a kernel parameter, and if during device discovery, the kernel determines that a device can address all of physical memory. Some devices can be in pass through mode without all devices being in this mode.

Because there is no permissions checking, our driver and microcode attacks work in pass through mode. Pass through mode is intended to use a software TLB [50], but we verified that on our system, the software TLB does not check permissions. In our system, even though GPU device addresses are 40 bits, it identifies as a 32-bit device during its initialization. Therefore, the kernel must boot with less than or equal to 4 GB of memory to enable pass through mode. We verified that regardless of how much physical memory is in the machine, if the kernel boots with a `mem=4G` option, the kernel defaults to pass through mode where our attacks work.

IOMMU deferred mode. If the IOMMU is enabled, Linux configures it to work in deferred mode by default. When system memory is unmapped from IO devices, the OS clears the IO page table entry and adds it to a flush list. The IOMMU driver flushes the entire IOTLB when the list contains a certain number of entries (250 for the kernel version 4.1) or at most 10ms after an entry is added to the list, whichever comes first.

Deferred mode is considered less secure for memory integrity because the memory unmapped from the device by the device driver remains accessible from the device for up to 10ms [26]. However, we show below that this mode foils the GPU malware attack.

IOMMU strict mode. Strict mode does not defer flushing the IOTLB when unmapping IO memory; each page is flushed by the driver immediately after its entry is unmapped from the IO page table. The driver flushes only the region covered by the entry (which can be a single page, or a power-of-two contiguous and aligned sequence of pages) and it never flushes the entire IOTLB, as it does in deferred mode.

[2]In Linux $t = 10$ for Intel IOMMUs

Workload	Bit rate	Stale period
Idle ssh connection	10 bps	1 day
Web radio	130 Kbps	1 hour
Web video: Auto (480p)	2 Mbps	1 minute

Table 2. Workload, average measured bit rate, and the time a stale IOTLB entry stays resident.

Strict mode also respects IO protection domains. Each device, e.g., NIC, GPU, is placed in its own protection domain and entries in the IOTLB are tagged with the domain identifier. When the IOMMU driver flushes an entry from one domain, it does not affect other domains.

This mode is considered safer than the deferred mode, because it flushes the IOTLB immediately after the entry gets unmapped.

4.3 IOMMU attacks

Our driver and microcode attacks work when the IOMMU is disabled or in pass through mode. Therefore, we now describe how we attack strict mode and how we can surreptitiously transition from deferred mode to strict mode.

We describe how to keep a stale malicious mapping in the IOTLB without having it installed in the I/O page table. Keeping the entry out of the I/O page table makes an attack more difficult to detect (though we know of no security tool that even validates I/O page tables).

Stale IOMMU entries in strict mode. Our attack writes a malicious IO page table entry (e.g., one that maps an unrelated process' credential structure), launches a GPU kernel which accesses the device address of the mapping, causing the entry to be cached in the IOTLB. Then the attack code overwrites the IO page table entry and the GPU kernel terminates. The malicious entry remains cached in the IOTLB, with no way for software to detect its presence, and no evidence of it in the backing IO page table.

The question remains, how long can a stale entry last in the IOTLB? We do experiments on a machine running Ubuntu 14.04 with X and Xfce desktop, the Linux 3.16 kernel with an Intel 82579LM Gigabit Network Card. We run several experiments each with progressively more network traffic to create different levels of IOTLB contention, because the network card competes for IOTLB entries with the GPU. We also tried to create IOTLB pressure from the GPU by running `glxgears`, a web browser, and displaying video, but none of these activities generated significant competition for IOTLB entries, and none of them caused the GPU or driver to flush the IOTLB.

Table 2 summarizes our experiments where we run workloads with increasing network load and measure the stale period—the period of time a stale entry stays in the IOTLB.

The first workload keeps an open `ssh` connection (using the `TCPKeepAlive` option) to an idle machine. We measure a stale period of more than 24 hours (after one day we discontinued the experiment). Next we continuously stream a web radio station which generates on average 130 Kbps of incoming network traffic. Without refreshing the stale entry, the stale mapping can be read after $T = 1$ hour. By periodically running an unprivileged GPU kernel to read the memory mapped by the stale IOTLB entry every hour, we can keep the stale IOTLB entry resident for 10 hours.

Finally, we increase the network load by streaming a video from youtube using the Firefox web browser. The video is played at the "Auto (480p)" setting, generating an average of 2 Mbps as measured by a network monitor. This workload uses the NIC and the graphics rendering capability of the GPU. The mapping can be reliably read 1 minute after it was erased from the IO page table. Running a GPU kernel every 1 minute is sufficient to keep the stale IOTLB entry resident for one hour, after which we discontinued the experiment. Streaming higher bandwidth videos, like the “Auto (720p)” setting, cause the attack to fail, even when we refresh the stale entry every minute. We use round numbers like 1 minute for a stale period to validate that our attacks are practical. Future work might determine more clever ways to keep an IOTLB entry resident, but our experiments establish a large enough window of vulnerability to be a security concern.

Stealthy transition from deferred to strict. Keeping a stale entry in the IOTLB is possible only if the IOMMU is configured in strict mode, because it is the only mode that invalidates each IOTLB entry separately. However, if the IOMMU is enabled, Linux uses deferred mode by default, which flushes the IOTLB as a whole. These IOTLB flushes frustrate our attacks (and form a practical countermeasure to both of our attacks).

We find that the kernel transitions between deferred and strict mode based on the state of a single variable (`intel_iommu_strict`). By setting this variable to 1, the kernel will put all devices into strict mode. Because this is a small, legal change to kernel state, it is quite stealthy. We experimentally verify that it is effective at engaging strict mode, where we can cache a stale IOTLB entry and launch one of the attacks we now describe.

We leave for future work developing a Linux IOMMU management policy that combines the best parts of strict and deferred mode. Strict mode minimizes the chances of memory corruption by a device by quickly unmapping DMA memory. Deferred mode frustrates our attacks by periodically flushing the entire IOTLB.

5 GPU driver attack

In this section we show an attack on the stock NVIDIA closed-source GPU driver. The attack enables arbitrary CPU memory mapping into an unprivileged GPU program, concealing malware code which monitors or changes CPU memory from a GPU kernel.

The attack scenario. An attacker loads a malicious kernel module which installs a backdoor by patching a GPU driver in CPU memory (Step ① in Figure 2). The driver continues to operate normally. To trigger the backdoor, an unprivileged GPU-controlling process performs a sequence of standard GPU API calls (a trigger sequence) ③, and maps the requested CPU memory into the GPU ④. The driver patch bypasses the standard access control checks in the driver, and allows the attacker to map any user or kernel memory page. The CPU process invokes a malicious, unprivileged GPU kernel ⑤ which accesses the mapped page ⑥. The attack module may unload itself, leaving the modified driver in kernel memory to subsequently repeat the attack from another unprivileged process. If no more attacks are planned, a stealthier alternative is to reconstruct the original driver code to evade detection by kernel code integrity scanners [35]. We implement two proof-of-concept GPU malware kernels – one that escalates the privileges of a given process by manipulating its `cred` structure, and another that diverts the execution flow of a given process by updating its code, which resides in read-only memory.

Our attack is similar to the previously reported GPU keylogger [24] attack. Both attacks exploit GPU DMA capabilities; in

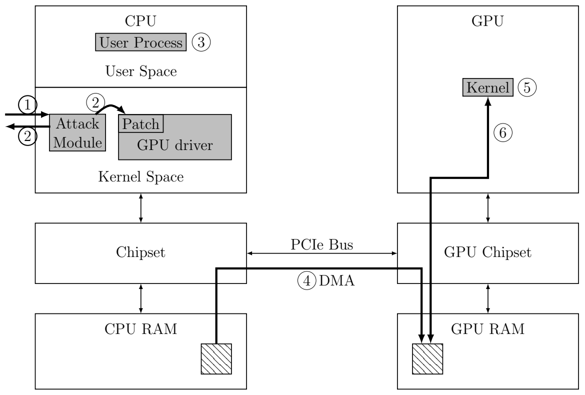

Figure 2. The driver attack, where a patched GPU driver has its memory mapping access control bypassed to allow a GPU kernel to access all of the CPU’s memory.

the keylogger case access is to the keyboard buffer in OS kernel memory. The primary difference is that our attack requires no long-running CPU process to proceed, while the keylogger does. That is because for the keylogger, the malicious memory mapping to the keyboard buffer must be installed into an unprivileged process that is running while the root-level compromise is active. The attack is lost if that unprivileged process subsequently terminates. Attacking the driver directly allows us to map any page to any process as many times as necessary, and without modifying sensitive kernel data structures.

Driver patch. Identifying the specific locations in the NVIDIA driver that control access to memory would seem difficult because most of the driver is proprietary and undisclosed. However, the particular functions that control memory mapping reside in the wrapper of the driver that is shipped open source and compiled at the time of driver installation, making it easier to determine the exact location that needs patching.

We choose to patch the NVIDIA driver in memory rather than modify its source code. Patching the driver enables the attack on a system where the GPU driver module has been already loaded and is in use by the kernel, allowing the attack to avoid unloading it first, which can be easily noticed. The patch is installed by a malicious module which finds the driver module in memory, overwrites some of its code, and then unloads itself.

The patch diverts the original control flow of the driver to bypass the memory permission checks when handling the `cudaHostRegister()` API call, which is normally used to lock CPU memory pages (`os_lock_user_pages()`) and map them into the GPU address space (`nv_dma_map_pages()`).

The driver acts normally as long as the trigger sequence has not been detected. The trigger sequence is a series of legal but erroneous calls to `cudaHostRegister()`, e.g., passing pointers to unallocated memory. Once triggered, the modified driver expects another call to `cudaHostRegister()` with the hidden buffer as a parameter. The hidden buffer contains the actual parameters of the malicious mapping request. For instance, the driver may map the virtual address of a certain process, or some known kernel data structure like the task control block (`task_struct` in Linux). The patch resolves the physical address of the requested memory region, injects this address into the original control structures of the driver, and resumes the original driver which updates the internal

GPU page table with the new mapping. The GPU-controlling process then may launch the malicious GPU kernel that accesses the mapped region.

6 GPU microcode attack

We demonstrate a novel attack that modifies GPU microcode to spy on or corrupt CPU and GPU state. The attack uses an embedded microprocessor in NVIDIA GPUs and can evade any detection mechanism that relies on evidence from CPU or GPU memory.

6.1 Background: NVIDIA GPU microprocessors

NVIDIA GPUs contain several on-board microprocessors for power management, video display, decoding, decryption, and other purposes. The existence of multiple Falcon microprocessors in NVIDIA GPUs has been officially disclosed by NVIDIA [32], but no official public API has been released. The reverse-engineering community discovered that Falcon microprocessors are capable of issuing data transfers from CPU or GPU physical memory to microprocessor memory using dedicated memory transfer instructions [13, 17].

Falcon microprocessors expose a common set of MMIO registers that both a CPU and the microprocessor can access or update (GPU kernels normally cannot access them). These registers enable communication between privileged CPU code and the microcode. Certain MMIO registers update the code and data memory of the microprocessor and restart its execution. Linux kernel source code contains the assembly code for certain Falcon processors[3].

The platform's GPU driver loads control code onto the Falcon microprocessors as part of its initialization sequence. The code is invoked in response to certain events, for example when serving requests to switch GPU control to another CPU process (called GPU context switch [15]). It is this control code that we attack. We call this a GPU microcode attack because the microprocessor code is one of several non-user visible code modules loaded into the GPU at its initialization, and because it is terminology accepted by GPU driver developers [16].

6.2 The attack

The attack consists of three phases: launch, monitor, and execute, as illustrated in Figure 3. In the launch phase, the attacker with privileged access installs the attack microcode on one of the Falcon microprocessors, and executes the code. Now the attack enters the monitor phase, in which the microprocessor monitors regions of GPU memory to identify commands from the attacker. The commands can be located in GPU memory or MMIO registers, though our prototype monitors only specific GPU memory locations. Finally, once the commands are identified, they are executed by the microprocessor as part of the execution phase.

In our proof-of-concept attack, we use a seemingly random binary string as the trigger for the attack. An unprivileged attacker executes a CUDA program that has a large data structure containing repeated copies of the trigger string. Our modified microcode detects the trigger (probably) in one of its monitoring locations and transitions into the attack execution phase. In our prototype, the attacking microcode escalates the privilege of a predefined running shell process, by writing into the process' credential structure in CPU memory.

Stealth and unobtrusiveness. The key benefit of this attack is stealth. We observe no evidence that the attack is occurring in

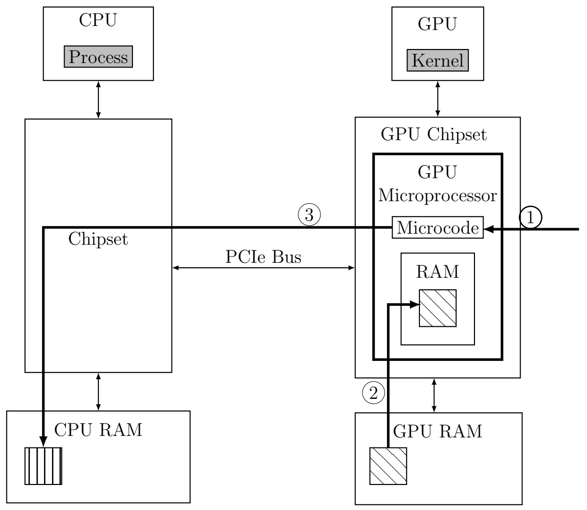

Figure 3. The GPU microcode attack. (① *Launch*) The attacker loads the attack code into microprocessor storage. (② *Monitor*) The microcode transfers data from GPU memory to its own memory in order to identify triggers or commands from the attacker. (③ *Execute*) Once it detects the commands, it launches the attack by writing to critical data structures in CPU memory.

CPU memory or in GPU memory. We check all of the GPU base address register (BAR) regions mapped by the CPUs (BAR0 for MMIO, BAR1 for VRAM aperture, BAR3 for kernel-accessible control memory) and none contain any sign of the microcode data or code, both in big- and little- endian representations. The only forensic tool we could find for dumping GPU memory [37] did not reveal the microcode. The microcode resides only in the GPU microprocessor memory. The Falcon processor has MMIO registers for uploading the microcode from CPU memory to microprocessor memory. It also has an MMIO register for transferring microprocessor memory back to CPU memory, but we know of no tool that uses this interface, let alone one that tries to determine if the microcode is malicious.

Once established, the microcode attack does not require support from a kernel module or a CPU user process, unlike previously known GPU malware attacks [24]. The attack does not affect the integrity of kernel-level data structures.

6.3 Technical details

The attack code is written in C, based on the microcode assembly code in the Linux kernel. It is then compiled by the publicly available LLVM backend and envytools [17]. The Falcon's `xfer` instruction can initiate DMA, and it can transfer data from both GPU and CPU memory to its own memory, or vice versa.

Out of many Falcon microprocessors, we use a microprocessor that manages context switching between multiple command streams, e.g., between the X server and a 3D application. There are several microprocessors involved in this process; microprocessors manage the context switch of a group of SMs (graphics processing clusters, or GPCs), and a HUB microprocessor that manages these GPC microprocessors. We update the microcode of HUB microprocessor because it has larger code and data memory, and the open-source version of the microcode is available in the Linux kernel. An official version of the microcode can be also extracted [16], and used to build modified microcode.

[3]Located in drivers/gpu/drm/nouveau/core/engine/graph/fuc.

The compiled binary is loaded into the microprocessor of the NVIDIA C2075 GPU using a set of MMIO registers. The `envy-tools` [13] project contains a set of tools the open-source community has developed and used to build the open-source `nouveau` Linux driver for NVIDIA GPUs. RNN is the community-built knowledge base for MMIO registers in NVIDIA GPUs. The addresses in parentheses are the offsets within the MMIO region, which is referred by the first set of the PCI base address registers (BAR0) of NVIDIA GPUs.

Microcode is uploaded to Falcon using `CODE_VIRT_ADDR` (0x188) and `CODE` (0x180) registers. The user can issue 32 bit stores to the `CODE_VIRT_ADDR` register for the index of the 256B chunk of code to be written, and `CODE` for the content of the code at the address pointed by `CODE_VIRT_ADDR`. `DATA` (0x1c4) allows upload of microcode data.

Most of the attack microcode is loaded into regions that do not overlap with the existing code or data, to have minimal effect on the normal system operation. Only small patches to the code region are needed to redirect the control flow to the injected attack code.

To remain unobtrusive, we split the work done by the attack microcode into small units and execute only for a limited time, chaining together subsequent executions using continuations. The microcode is originally designed to be interrupt-driven: most of the functions are interrupt handlers for different types of interrupts, such as periodic timers or commands waiting to be handled. The microprocessor assumes that interrupt handlers will be brief.

We test the unobtrusiveness of our attack code by running `glxgears` from the Mesa GL utility library. This application reports the frame rate of 3D graphics rendering, and we could observe lower frame rates or even GPU lockup if the execution time of our inserted operations took too long. By fine-tuning the amount of work done at each interrupt, our attack microcode supports the same `glxgears` framerate as the unmodified microcode. We also could not subjectively observe an effect on typical desktop operations.

We implement our proof-of-concept microcode attack on top of the open source nouveau driver. The attack code for the nouveau driver includes all of the attack steps we describe in the attack scenario. We verify that we can unobtrusively inject simple code sequences into the NVIDIA microcode, but do not implement the entire attack for the NVIDIA microcode.

The embedded NVIDIA microprocessor has a periodic timer interrupt and a one-shot watchdog timer, but in our experience, the use of the periodic timer affects the graphics output. Therefore, to remain undetectable, we use the watchdog timer and have the watchdog event handler reschedule another watchdog event. Our attack code is 4KB, which we add to the 3KB of Nouveau microcode fitting comfortably in the device's 16KB capacity.

6.4 Discussion

Falcon microprocessors are relatively slow, the one we used runs at 270MHz. We only read a small number of GPU memory locations to keep execution time short. Therefore, our trigger consists of a GPU program that fills much of its memory with the target string which gives a high probability of it being read by the attack microcode. Our proof-of-concept trigger fills 3GB of data and the microcode reads five memory locations at 1GB offsets. This combination makes the microcode recognize the trigger for each of 10 trials.

Small code size. Different types of Falcon processors have different limits on code and data memory. For example, the maximum code size and data size for a microprocessor are 16 KB and 4 KB, respectively, whereas the limits for closely related microprocessors is only 8 KB and 2KB. If multiple microprocessors communicate and launch a more complex attack than one microprocessor can handle, the attacker can distribute the work according to the memory limit of each processor.

Microcode validation. Starting from Maxwell GPUs, NVIDIA significantly strengthened the security of Falcon microprocessors by requiring their code to be signed and preventing code modifications after code is initially loaded [32]. Unsigned microprocessor code may run in unsecure mode, but it cannot use certain hardware features (the precise set of constraints depends on the processor). These new security mechanisms, therefore, are likely to complicate or even entirely prevent our microcode attack, because most (but not all) Falcons on NVIDIA Pascal GPUs do not allow unsigned code access to physical memory. We leave the vulnerability analysis of Pascal GPUs for future work.

7 Related work

GPU malware. Vasiliadis et al. [45] present two GPU malware techniques, code unpacking and runtime polymorphism, used to evade malware detection. These techniques make use of the GPU computing capacity to build more complex packing algorithms and leverage GPU direct memory access (DMA) to modify host memory.

Ladakis et. al. implement a keylogger on the GPU [24], leveraging the DMA capability of GPUs to monitor the operating system keyboard buffer from a GPU kernel. The GPU-based keylogger requires an unprivileged helper process to set up the attack. It relies on a kernel module to update a page table entry of the helper, so that the process' address space contains a window on the kernel-level keyboard buffer. The keyboard buffer address is then moved to the GPU page table, and erased from the page table in the CPU, keeping the kernel memory mapping for a short time.

Both of these attacks require helper processes on the CPU, and these processes violate certain address space integrity properties (though in most systems, these integrity properties are implicit). Hiding malware with unpacking and polymorphism on the GPU requires mapping a CPU memory region that is executable, writable, and IO-mapped. The GPU-based keylogger has a user-level page that maps kernel memory which no user process should ever map. These distinctive memory regions that clearly violate certain safety properties make the malware easy to detect by some rootkit detectors [19, 36].

The GPU-based microcode attack described in this paper does not require any running process once it is installed in the microprocessor. It leaves no trace in CPU or GPU memory and therefore does not violate any memory integrity property. The GPU driver attack does not need a CPU helper until the malicious behavior is triggered. The attack is entirely encapsulated in the driver and does not change any kernel data structure, however the patched driver module might still be detected by kernel integrity checkers.

Villani et. al. analyze four GPU-assisted malware anti-memory-forensics techniques without modification of GPU microcode: unlimited code execution, process-less code execution, context-less code execution and inconsistent memory mapping and apply them to integrated Intel GPU cards [46].

GPU as secure co-processor. PixelVault [44] proposes to use GPUs as secure co-processors for cryptographic operations. We have shown in this paper how features from the official NVIDIA debugger to unofficial hardware interfaces violate the security assumptions of PixelVault.

Firmware attacks. There are several firmware-based attacks that target diverse devices [1, 3, 10, 41, 47, 51]. Similar to the microcode attack in this paper, these attacks embed malicious code into the firmware to circumvent the platform's security while evading detection. Triulzi [42] presents a sniffer that uses a combination of NIC and GPU to access main memory. The GPU runs an ssh daemon that accepts packets from the NIC through PCI-to-PCI transfer. The firmware modification on the GPU is mainly due to lack of PCI-to-PCI transfer support. With GPUDirectRDMA [31], this attack can be implemented without GPU firmware modification. To the best of our knowledge, our attack is the first GPU microcode-based attack that leverages GPU embedded microprocessors.

Newer NVIDIA GPUs are expected to disallow the use of unsigned microcode, preventing the microcode attack.

Information leaks through GPU. Recent works notice that the GPU driver does not erase the device memory after kernel termination, leaking private information [25, 28]. This paper describes a different type of attacks that leverage the GPU to stealthily perform unauthorized accesses to CPU memory.

Attacks using graphics software stack. The security aspects of using GPUs in graphics applications have been the subject of much work [38, 39, 43]. Our work is complementary in that we focus on the GPU microcode and the driver, and investigate the weaknesses of using GPUs as secure co-processors.

Reverse-engineering GPU hardware. Detailed information about GPU hardware architecture is usually never disclosed by the vendors. Wong et al. [48] reverse engineer GPU internals via carefully crafted microbenchmarks. We use similar techniques in this paper to discover the size of the instruction cache. Fujii et al. [17] explain the internal organization of GPU microprocessors which we use to implement the microcode attack.

8 Conclusion

GPUs are not an appropriate choice for a secure coprocessor, and they pose a security threat to computing platforms, even those with an IOMMU. The problem with making hardware, especially hardware as complex as a GPU, into something that enhances security is that the security guarantees rely on a large set of assumptions about architectural, micro-architectural, and software features. These assumptions are difficult to verify and they can change for different versions of the product because the underlying motivation of the manufacturer is not security.

As an attack platform, they combine powerful access to platform hardware with an opacity encouraged by its proprietary nature. While forensic tools for GPUs will improve, they represent another nettlesome resource for determined attackers.

9 Acknowledgements

Mark Silberstein was supported by the Israel Science Foundation (grant No. 1138/14) and the Israeli Ministry of Science. We also gratefully acknowledge funding from NSF grants CNS-1017785 and CCF-1333594.

References

[1] Brocker, M., and Checkoway, S. iSeeYou: disabling the MacBook webcam indicator LED. In *Proceedings of the USENIX Security Symposium* (2014), USENIX Association, pp. 337–352.

[2] Corp., I. *Intel 965 Express Chipset Family and Intel G35 Express Chipset Graphics Controller Programmers Reference Manual. Volume 1: Graphics Core*, 2008. https://01.org/sites/default/files/documentation/965_g35_vol_1_graphics_core_0.pdf.

[3] Cui, A., Costello, M., and Stolfo, S. J. When firmware modifications attack: A case study of embedded exploitation. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)* (2013).

[4] CVE. CVE-2007-1019. https://www.cvedetails.com/cve/CVE-2007-1881/. Accessed: May 2016.

[5] CVE. CVE-2011-1019. https://www.cvedetails.com/cve/CVE-2011-1019/. Accessed: May 2016.

[6] CVE. CVE-2014-5207. https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-5207. Accessed: May 2016.

[7] CVE. CVE request: ro bind mount bypass using user namespaces. http://www.openwall.com/lists/oss-security/2014/08/13/9. Accessed: May 2016.

[8] CVE. How to exploit the x32 recvmmsg() kernel vulnerability CVE 2014-0038. http://blog.includesecurity.com/2014/03/exploit-CVE-2014-0038-x32-recvmmsg-kernel-vulnerablity.html. Accessed: May 2016.

[9] CVE. Local root exploit for CVE-2014. https://github.com/saelo/cve-2014-0038. Accessed: May 2016.

[10] Duflot, L., Perez, Y.-A., and Morin, B. What if you can't trust your network card? In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)* (Berlin, Heidelberg, 2011), Springer-Verlag, pp. 378–397.

[11] Duflot, L., Perez, Y.-A., Valadon, G., and Levillain, O. Can you still trust your network card. *CanSecWest/core10* (2010).

[12] Edge, J. A crypto module loading vulnerability. https://lwn.net/Articles/630762/. Accessed: May 2016.

[13] envytools. envytools - tools for people envious of nvidia's blob driver. https://github.com/envytools/envytools. Accessed: May 2016.

[14] Fischer, W. Activating the Intel VT-d virtualization feature. https://www.thomas-krenn.com/en/wiki/Activating_the_Intel_VT-d_Virtualization_Feature. Accessed: May 2016.

[15] freedesktop.org. nouveau context switching. http://nouveau.freedesktop.org/wiki/ContextSwitching/. Accessed: May 2016.

[16] freedesktop.org. nouveau context switching firmware. http://nouveau.freedesktop.org/wiki/NVC0_Firmware/. Accessed: May 2016.

[17] Fujii, Y., Azumi, T., Nishio, N., Kato, S., and Edahiro, M. Data transfer matters for GPU computing. In *Proceedings of the International Conference on Parallel and Distributed Systems (ICPADS)* (Washington, DC, USA, 2013), IEEE Computer Society, pp. 275–282.

[18] Gerfin, G., and Venkataraman, V. Debugging experience with CUDA-GDB and CUDA-MEMCHECK. In *GPU Technology Conference* (2012).

[19] Hofmann, O. S., Dunn, A., Kim, S., Roy, I., and Witchel, E. Ensuring operating system kernel integrity with OSck. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (March 2011).

[20] Hofmann, O. S., Porter, D. E., Rossbach, C. J., Ramadan, H. E., and Witchel, E. Solving difficult HTM problems without difficult hardware. In *Proceedings of the 2nd Workshop on Transactional Computing (TRANSACT)* (Portland, OR, August 2007).

[21] Kato, S. Gdev: Open-source GPGPU runtime and driver software. https://github.com/shinpei0208/gdev.

[22] Kato, S., McThrow, M., and Maltzahn, Carlosand Brandt, S. Gdev: First-class GPU resource management in the operating system. In *Proceedings of the USENIX Annual Technical Conference* (June 2012).

[23] Khronos Group. *The OpenCL Specification, Version 2.0*, 2014.

[24] Koromilas, L., Vasiliadis, G., Ioannidis, S., Ladakis, E., and Polychronakis, M. You can type, but you can't hide: A stealthy GPU-based keylogger. In *Proceedings of the Sixth European Workshop on System Security (EUROSEC)* (2013), ACM.

[25] Lee, S., Kim, Y., Kim, J., and Kim, J. Stealing webpages rendered on your browser by exploiting GPU vulnerabilities. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)* (Washington, DC, USA, 2014), IEEE Computer Society, pp. 19–33.

[26] Malka, M., Amit, N., Ben-Yehuda, M., and Tsafrir, D. rIOMMU: Efficient IOMMU for I/O Devices That Employ Ring Buffers. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2015), ASPLOS '15, ACM.

[27] Mareck, R. *AMD x86 Firmware Analysis*. Accessed: May 2016.

[28] Maurice, C., Neumann, C., Heen, O., and Francillon, A. Confidentiality issues on a GPU in a virtualized environment. In *Financial Cryptography and Data Security*. Springer, 2014, pp. 119–135.

[29] nouveau. Nouveau: Accelerated open source driver for nvidia cards. http://nouveau.freedesktop.org/wiki/. Accessed: May 2016.

[30] NVIDIA. *Debugger API*. http://docs.nvidia.com/cuda/debugger-api/index.html.

[31] NVIDIA. *GPUDirectRDMA technology*. http://docs.nvidia.com/cuda/gpudirect-rdma/index.html.

[32] NVIDIA. NVIDIA Falcon security. ftp://download.nvidia.com/open-gpu-doc/Falcon-Security/1/Falcon-Security.html.

[33] NVIDIA. *CUDA Runtime API*, 2014.

[34] OLEKSIUK, D. Breaking uefi security with software dma attacks. http://blog.cr4.sh/2015/09/breaking-uefi-security-with-software.html. Accessed: May 2016.

[35] PENDERGRASS, J. A., AND N. MCGILL, K. LKIM: The linux kernel integrity measurer. In *Johns Hopkins APL Technical Digest*. September 2013.

[36] PETRONI, JR., N. L., AND HICKS, M. Automated detection of persistent kernel control-flow attacks. In *Proceedings of the 14th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2007), CCS '07, ACM, pp. 103–115.

[37] RICHARD III, G. GPU memory dump tool. 2015 DFRWS Forensics Challenge, 2015. http://cs.uno.edu/~golden/Materials/gpumalware/nvidia-dump-fb-1.0.tar.gz.

[38] SECURITY, C. I. WebGL-a new dimension for browser exploitation. http://www.contextis.com/resources/blog/webgl-new-dimension-browser-exploitation/. Accessed: 2015-02-15.

[39] SECURITY, C. I. WebGL-more WebGL security flaws. http://www.contextis.com/resources/blog/webgl-more-webgl-security-flaws/. Accessed: May 2016.

[40] SEVINSKY, R. Funderbolt: Adventures in thunderbolt dma attacks. https://media.blackhat.com/us-13/US-13-Sevinsky-Funderbolt-Adventures-in-Thunderbolt-DMA-Attacks-Slides.pdf. Accessed: May 2016.

[41] STEWIN, P. *Detecting peripheral-based attacks on the host memory*. PhD thesis, Technische Universität Berlin, 2015.

[42] TRIULZI, A. Project maux mk.ii. http://www.alchemistowl.org/arrigo/Papers/Arrigo-Triulzi-PACSEC08-Project-Maux-II.pdf. Accessed: May 2016.

[43] US-CERT. Vulnerability summary for CVE-2014-8298. National Vulnerability Database, Dec 2014. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-8298.

[44] VASILIADIS, G., ATHANASOPOULOS, E., POLYCHRONAKIS, M., AND IOANNIDIS, S. PixelVault: Using GPUs for securing cryptographic operations. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2014), ACM.

[45] VASILIADIS, G., POLYCHRONAKIS, M., AND IOANNIDIS, S. GPU-assisted malware. *International Journal of Information Security* (2010), 1–9.

[46] VILLANI, A., BALZAROTTI, D., AND DI PIETRO, R. The impact of GPU-assisted malware on memory forensics: A case study. In *DFRWS 2015, Annual Digital Forensics Research Conference, Philadelphia, USA* (2015).

[47] WOJTCZUK, R., AND RUTKOWSKA, J. Attacking Intel trusted execution technology. *Black Hat DC* (2009).

[48] WONG, H., PAPADOPOULOU, M.-M., SADOOGHI-ALVANDI, M., AND MOSHOVOS, A. Demystifying GPU microarchitecture through microbenchmarking. In *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on* (March 2010), pp. 235–246.

[49] YASSOUR, B.-A., BEN-YEHUDA, M., AND WASSERMAN, O. On the DMA mapping problem in direct device assignment. In *Proceedings of the 3rd Annual Haifa Experimental Systems Conference* (New York, NY, USA, 2010), SYSTOR, ACM, pp. 18:1–18:12.

[50] YU, F. Intel IOMMU Pass Through Support. https://lwn.net/Articles/329174/. Accessed: May 2016.

[51] ZADDACH, J., KURMUS, A., BALZAROTTI, D., BLASS, E.-O., FRANCILLON, A., GOODSPEED, T., GUPTA, M., AND KOLTSIDAS, I. Implementation and implications of a stealth hard-drive backdoor. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)* (2013), ACM, pp. 279–288.