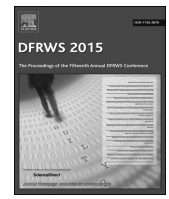
Digital Investigation 14 (2015) S16–S24

Contents lists available at ScienceDirect

Digital Investigation

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



DFRWS 2015 US

The impact of GPU-assisted malware on memory forensics: A case study

Davide Balzarotti [a], Roberto Di Pietro [b, 1], Antonio Villani [c, *]

[a] Eurecom, Sophia Antipolis, France
[b] Bell Labs, Paris, France
[c] Department of Mathematics and Physics, University of Roma Tre, Rome, Italy

Keywords:
Graphic processing units
Memory analysis
Malware
Digital Forensics
Direct Rendering Manager

A B S T R A C T

In this paper we assess the impact of GPU-assisted malware on memory forensics. In particular, we first introduce four different techniques that malware can adopt to hide its presence. We then present a case study on a very popular family of Intel GPUs, and we analyze in which cases the forensic analysis can be performed using only the host's memory and in which cases it requires access to the GPU's memory. Our analysis shows that, by offloading some computation to the GPUs, it is possible to successfully hide some malicious behavior. Furthermore, we provide suggestions and insights about which artifacts could be used to detect the presence of GPU-assisted malware.

Introduction

For more than twenty years computers have relied on dedicated Graphics Processing Units (GPUs) to perform graphical computations and rendering. More recently, with the advent of general-purpose computing on graphics processing units (GPGPU), GPUs have been increasingly adopted also to perform other generic tasks. In fact, thanks to their many-core architecture, GPUs can provide a significant speed-up for several applications – such as financial and scientific computations, regular expression matching, bitcoins mining, video transcoding, and password-cracking.

Despite their pervasiveness and their ability to perform generic computations, the role and impact of GPUs to perform malicious activities is still largely understudied. To the best of our knowledge, no GPU-assisted malware has been discovered in the wild – with the sole exception of BadMiner (ArsTechnica, 2015) which exploits the presence of a GPU to mine bitcoins. However, researchers have already shown that several malicious activities can take advantage of GPUs, for instance to steal sensitive information (Kotcher et al., 2013; Lee et al., 2014), unpack malicious code (Vasiliadis et al., Oct 2010), and hide malicious activities from malware detection and analysis tools (Triulzi, 2008; Ladakis et al., 2013). It is important to notice that these threats are not confined to the graphic environment (e.g., screen grabbing or unrestricted access to GPU memory) but that GPUs can also be abused to perform attacks that are outside the graphic domain. In fact, opposite to others commonly available PCI devices, the fact that GPUs can be programmed by userspace applications makes them a very attractive attack vector. As a matter of fact, also other peripherals (such as hard disks, network devices, printers) have been abused to perform malicious activities (Zaddach et al., 2013; Sparks et al., 2009; Cui et al., 2013), but those scenarios required to introduce custom modifications to the device firmware. GPUs are instead meant to be programmed by the end-user and require no firmware

* Corresponding author.
E-mail addresses: davide.balzarotti@eurecom.fr (D. Balzarotti), roberto.di_pietro@alcatel-lucent.com (R. Di Pietro), villani@mat.uniroma3.it (A. Villani).

[1] He is also with the Department of Maths, University of Padua, Padua, Italy.

http://dx.doi.org/10.1016/j.diin.2015.05.010

modification to execute arbitrary code. To make things worse, graphic cards vendors pay more attention to the performance of their products than to their security. Unfortunately, this has often a negative impact on several security mechanisms, such as the memory isolation between independent process running on the same GPU (Di Pietro et al., 2013; Maurice et al., 2014).

From a defense perspective, the main problem is that neither antivirus softwares nor memory forensics tools are currently able to analyze the content of the GPU memory. However, it is not clear how severe this limitation is in practice, and what is the real level of stealthiness that can be achieved by GPU-assisted malware. In particular, the main goal of this paper is to understand what is the impact of GPU-assisted malware on memory forensics. Are traditional *collection* and *analysis* techniques sufficient to detect and fully understand the malicious behavior? This is a fundamental question for the computer forensic domain. Nowadays, memory forensics tools have an incomplete view of the system, limited only to the content of the operating system memory. It is very important to understand under which conditions this piece of information is enough to detect the presence of a malicious activity (even if the malware relies to a certain extent on the use of the GPU). Only recently GPUs gained the attention of the digital forensic community and a common assumption in the current research (Richard, 2015) is that GPU-assisted malware needs at least a component running in the system memory. If so, the analysis of this component could be sufficient to detect that the malware is using the GPU and maybe even to retrieve the purpose of the outsourced computation.

However, in this paper we show that in certain conditions it is possible for an attacker to leave no trace in the OS memory. In this case, malware coders have a substantial advantage that could only be limited by developing custom techniques to acquire and analyze the internal GPU memory of each vendor (as well as graphic card model, and device driver). Unfortunately, this can be a very difficult and time-demanding task – especially for proprietary products for which no documentation exists about their internal data structures.

Our goal is to answer a number of very practical questions. For instance, is it possible for an analyst to identify which processes are using the GPU and which code is being executed there? To answer the previous question, is it enough to analyze an image of the system memory, or does the analyst need also to inspect the regions memory-mapped to the video card (potentially containing proprietary data structures) or, worst case, does the internal memory of the video card also need to be collected and analyzed? Which of these questions can be answered by only looking at the system memory? Can the GPU be used to implement anti-forensic techniques? To address these questions, we designed and implemented a number of tests and experiments, based on a popular family of Intel GPUs.

To summarize, this paper makes the following contributions:

- We conduct the first study of the impact of GPU-assisted malware on memory forensics. In particular, we introduce four possible attack scenarios and discuss their impact for a forensic analyst. For each scenario we implement a different proof-of-concept for Linux and we report under what conditions the memory analysis can be performed through the kernel data structures, the video driver or the GPU memory
- We present a detailed case study on Intel integrated GPUs, which are part of all modern processors. Our results show that an adversary can take advantage of these integrated GPUs to obtain full access to any memory page in the host. Even worse, we show how it is possible to obtain a persistent malicious code running on the GPU without any associated process running on the host.

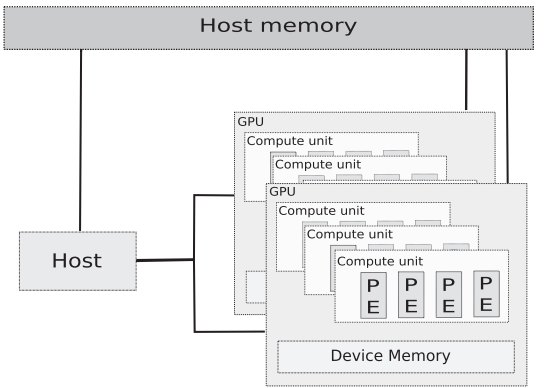


Fig. 1. A simplified view of a GPU-based heterogeneous architecture. The GPUs have access both to a dedicated device memory and to the host memory.

Background

Modern GPUs are specialized massively parallel processors with hundreds of computational units. In fact, while traditional CPUs dedicate most of their die area to cache memory, GPUs dedicate it to logic circuits. The combination of the two processing units, which takes advantage of the CPU for general applications tasks and of the GPU for highly parallel computation, is today the most common example of a heterogeneous architecture (see Fig. 1 for a simplified example). The host, namely the logical element containing the CPU, has its own memory and coordinates the execution of one or more GPUs in a master-slave configuration. The GPUs, which also have their own local memory, consist of several identical computational units, each containing the same amount of processing elements. A task which is executed on a GPU is called *kernel*. The GPU drivers manages *kernels* through the means of contexts, the equivalent of a process control block in the GPU.[2] A key aspect of this architecture is that the GPU is also able to directly access arbitrary pages on the host

[2] To disambiguate between the operating system kernel and the GPU *kernel*, the second one will be typed in italic.

memory. One of the motivation for this is that, in order to coordinate the executions of tasks on the GPU, the CPU typically writes commands into a *command buffer* that is consumed by the GPU. The same execution model is adopted by both graphic and general purpose tasks. Indeed, regardless on which high level library is used (e.g. OpenGL, CUDA, or OpenCL), eventually all commands are appended into the *command buffer*.

Physical memory acquisition

The dump of the physical memory allows a forensic investigator to extract detailed information about the system state, its configuration, and possible malware infections. However, the amount of useful information that can be extracted from a memory dump strictly depends on the way the memory image is obtained. Modern open-source memory acquisition tools for Linux (such as *LiMe* (LiMe, 2015) and *pmem* (Rekall, 2015)) load a kernel module and perform the acquisition from kernel space. Both LiMe and pmem automatically select the address ranges that are associated to the main system memory by examining the *iomem_resource* linked list in the Linux kernel and selecting only the address ranges marked as *System RAM*. This is necessary because the access to other memory regions can have side effects due to the presence of memory mapped I/O (MMIO), potentially resulting in a system crash (Stttgen & Cohen, 2013).

A portion of the MMIO memory is reserved to the PCI bus. At boot time, the BIOS instructs the operating system on which memory areas should be used for this purpose. Therefore, the CPU and the DRAM controller have two different views of the memory. The main difference is that the DRAM controller does not see memory ranges consumed for MMIO. The TOLUD register marks the top of the low (i.e. below 4 GB) usable physical memory that is accessible to the CPU. The CPU views the memory range from TOLUD to 4 GB as reserved for MMIO, while the DRAM controller views the same range to be allocated to DRAM.

All the memory areas assigned to the PCI subsystem are marked as *Reserved* and they are dumped neither by LiMe nor by pmem.

Threat model

In this paper we use the term *GPU assisted malware* (or just GPU malware for simplicity) to describe malicious software that performs some of its computation in the GPU. This definition covers the case in which the malware delegates some tasks to the GPU for pure performance reasons (e.g., to mine bitcoins in a more efficient way) and the case, more interesting for our study, in which the malware leverages the GPU for anti-forensic purposes.

However, in this paper we do not consider neither malicious hardware, nor code that requires the modification of the graphic card's firmware. A considerable effort has already been dedicated to study firmware-level malware in several contexts, spanning from chipsets, hard-drives, and graphic cards (Zaddach et al., 2013; Sparks et al., 2009; Cui et al., 2013; Triulzi, 2008). Current forensic analysis techniques are often ineffective in this scenario, and a more general approach (independent from the device that has been compromised) is needed to handle these powerful cases.

Finally, our definition of GPU assisted malware does not cover attacks against the GPU. For instance, this includes malicious code which resides completely in the host memory but which has the GPU memory as main *target*, e.g., to retrieve sensitive data from the content of the framebuffer or from the graphic buffer objects allocated by the GPU driver.

GPU assisted malware

The goal of our paper is to understand what is the impact of gpu-assisted malware on memory forensics. For this reason, it is convenient to classify each type of GPU malware according to two sets of requirements: the operating system privileges required by the malicious sample, and the amount of internal information about the graphic card required to realize the malware itself. According to these characteristics, we identify three main classes: user-space GPU malware, super-user GPU malware, and kernel-level GPU malware.

Userspace malware includes all the GPU malware samples that do not require administrative privileges on the victim host. This kind of malware simply exploits the possibility to execute general purpose computation on the GPU and, therefore, from an OS perspective it only uses legitimate APIs and it does not rely on any underlying software bug. For instance, Ladakis et al. (2013) presented an example of this category that uses the GPU to execute the unpacking routine of a generic malware.

Super-user malware requires instead administrative privileges on the target host. This allows the malicious code to perform a number of additional operations that are not available from user space. However, in this case we assume that the malware writer only relies on well defined libraries and APIs – without tampering with the internal information used by the graphic driver or the card itself.

The most powerful case is represented by kernel-level malware. In this case, on top of having administrative privileges on the host, the malware also knows the internal implementation of the graphic driver. As a result, it is able to modify arbitrary data structures in kernel space performing Direct Kernel Object Manipulation (DKOM). While from a strict operating system perspective, super-user and kernel access are equivalent, we prefer to separate these two scenario because they may have different consequences on the portability of the malicious code and on the forensic examination itself.

GPU malware and memory forensic

In presence of GPU malware, a forensic analyst needs to be able to answer a certain number of questions. In particular, in this paper we focus on three of them, which we believe are the most important for an investigation: (1) the enumeration of the processes that are using the GPU, (2) the extraction of the code that those processes have been executing in the video card, and (3) the identification

of the memory areas that the code in the video card has access to.

To answer these three questions, the analyst needs to collect a memory dump and to extract a certain number of information. Our goal is not to present a step-by-step description of how to perform this procedure. Instead, we want to study which of these three questions can be answered depending on the type of memory acquired, the class of GPU malware (userspace, super-user, or kernel-level), and the knowledge of the analyst (limited to the Operating system internals, or covering also the internals of the video card manufacturer). We stress that we are interested on the impact on memory forensic of GPU malware. Section "Related Work" provides a brief description on what malicious tasks a GPU malware can execute and how it can achieve persistence.

Anti-forensic techniques

The operating system, some of its core services (e.g., X windowing system), and/or the graphic driver usually maintain a number of internal data structures that describe who is using the graphic card and which tasks have been scheduled for execution. Since all of this data resides in the system memory, it may seem that a properly designed memory forensic tool should be able to answer all our three questions without the need to access the GPU memory.

In the next section we will see how this is indeed possible in most of the cases. However, malware developers do not need to always play by the rules, and therefore we investigate a number of possible anti-forensic techniques that could be used to reduce the footprint in the system memory.

Unlimited code execution

Under normal conditions, executing code on the GPU requires a controlling process running on the host. The host process adds a task on the command queue, which will be eventually fetched and executed by the GPU. However, GPUs have a non-preemptive nature: once the execution of a task is initiated, the GPU is locked with the execution of that task and no one else can use the GPU in the meanwhile. This is particularly problematic when the GPU is used both for rendering and computation, as this could generate undesired effects such as an unresponsive user interface.

As a consequence, in order to ensure a proper behavior, the graphic driver usually enforces a timeout to kill long lasting *kernels*. For GPU malware this could represent an important limitation because the malicious *kernel* needs to be sent over and over in a loop, making it more easy to detect in system memory.

The first anti-forensic technique consists in disabling the existing timeout to take full control of the GPU. For instance, in Vasiliadis et al. (2014) the authors disabled the *GPU hangcheck* to lock the GPUs indefinitely.

Process-less code execution

The GPU execution model involves the presence of a host process *P* controlling a GPU *kernel K*. This is an advantage for a forensic investigation, since the operating system should always maintain a link between a task executed in the GPU and the process which is responsible for that execution. However, in Section "Case study: Intel GPUs Architecture" it will be shown how the GPU execution model can be broken on Intel GPUs allowing the presence of a running *kernel K* without any controlling process *P*. In such case, the presence of the malware could still be detected by looking at the GPU driver itself. In fact, an analysis of the memory of the driver (which resides in the host memory) would reveal the presence of a GPU context running a *kernel* K without any controlling process P.

Context-less code execution

In the previous point we say how the graphic drivers stores information about the task being executed on the GPU (refer to Section "Case study: Intel GPUs Architecture" for more details). A more advanced anti-forensic technique could directly target the kernel objects to detach the *kernel* from the list of contexts in the GPU driver and remove any trace regarding the existence of a particular GPU *kernel* (of course, this only makes sense if the malware already achieves unlimited code execution). If context-less code execution is possible, and we will investigate that in the next section, then the GPU malware can completely hide his presence from the host memory.

Inconsistent memory mapping

In (Vasiliadis et al., Oct 2010), the authors propose to use the GPU to implement a stealthy keylogger that runs inside the graphic card and accesses through DMA the physical page containing the keyboard buffer. This makes the detection of the keylogger functionality more difficult, as the keyboard buffer does not appear in the list of memory pages mapped by the malicious process. In our tests we discovered that the list of accessible pages is kept both in the operating system and in the GPU memory. However, we will see how the two page tables do not necessarily need to contain the same information. This technique has some similarities with the TLB desynchronization attack (Sparks & Butler, 2005b); however, in our case a custom page fault handler is not required.

Case study: Intel GPUs architecture

GPUs can be broadly divided into two categories: (i) *discrete* GPUs with dedicated device memory and, (ii) *integrated* GPUs that use a portion of DRAM for graphics. For our analysis, we focus on the integrated GPU of Intel, namely the *Intel Integrated Graphic Devices* (IGDs). However, the concepts and ideas described in this paper can be also applied to other hardware and software configurations. In particular, in Section "Beyond our Case Study" we will discuss how our findings can be extended beyond our case study.

It is important to note that our study focuses on the Linux Direct Rendering Infrastructure (DRI), a framework to allow userspace applications to access graphics hardware (Freedesktop, 2015). In DRI, a userspace application can talk to the graphic drivers in two ways: directly using ioctls or indirectly through the X server. As it will be described in

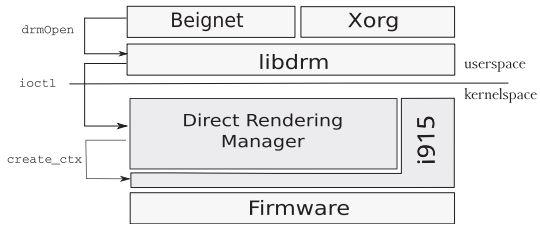


Fig. 2. Overview on the Linux Beignet architecture.

the following, OpenCL applications uses ioctls to interact with the GPUs. Therefore, in this work we focus on the former case. Nonetheless, in a different context the X server memory could also contain useful artifacts and therefore should also be analyzed to extract forensics information.

Fig. 2 shows a simplified view of the reference software stack. Note that we represented only the DRI's software components which are involved in our study. In the following, we describe briefly each layer. The information contained in the current section has been obtained from several sources including the source code of i915.ko, the official documentation of Intel (Intel, 2015) and unofficial sources such as (Vetter, 2015).

The userspace components: Beignet and libdrm

The Open Computing Language (OpenCL) is an open standard for parallel programming of heterogeneous systems. OpenCL allows the parallel programming of modern processors found in personal computers, servers and mobile devices. Beignet (2015) is an open source OpenCL implementation for Linux supporting two Intel processors: Ivy-Bridge and Haswell.

Beignet is composed by two main components: the OpenCL runtime and the Just-in-Time compiler. The runtime consists of a dynamic library which implements the OpenCL API. The Just-in-Time (JIT) compiler uses LLVM to implement the OpenCL C language compiler and to translate the code of the *kernel* into the ISA of the IGD. The *kernel* code is compiled through JIT because the OpenCL programs should be device-independent and could be executed on any hardware platform, supported by OpenCL.

In order to implement the runtime, Beignet uses a user-space library called libdrm. Libdrm is a cross-driver middleware which allows user-space applications to communicate with the kernel. The library provides wrapper functions for the ioctls to avoid exposing the kernel interface directly. Libdrm is a low-level library, typically used by graphics drivers such as the X drivers, and provides a common API for higher layers and implements specific functions for several vendors such as Intel, NVIDIA and AMD. In this paper we will rely on the Intel functions.

All our tests were conducted on an Intel Haswell processor with Beignet 0.9.1 and libdrm 2.4.58.

Direct rendering manager

The Direct Rendering Manager (DRM) is a subsystem of the Linux kernel responsible for the interface with the GPU. The DRM exposes an API that user space programs can use to send commands and data to the GPU, and perform operations like memory allocation on the GPU memory. Graphics drivers may also use DRM functions to provide a uniform interface to applications for memory management, interrupt handling and DMA. While user-space applications can interact directly with the GPU through ioctls, in many cases developers use the more high-level interface provided by the libdrm library.

The DRM supports two memory managers: the Translation Table Maps (TTM) and the Graphics Execution Manager (GEM). In this paper we focus on the GEM, which is used by the Intel graphic driver. GEM is designed to manage both the graphic memory and the graphic execution contexts, allowing multiple host processes to share graphics device resources. However, GEM only provides generic functions and each vendor implements its own functionality to support the GEM interface (in our case study the GEM functions were implemented inside the Intel graphic driver). The buffers allocated from GEM are called *buffer objects*.

The Intel graphic driver

The Intel graphic driver, called i915 in the main branch of the Linux kernel, implements a superset of the ioctls defined in the DRM layer (we used the version 3.14 of the Linux kernel in our experiments). The IGD is controlled by the CPU in two ways: through a set of memory-mapped *IO registers*, and through the *ringbuffers*, a set of queues containing the list of commands to be executed on the IGD. In particular, in Haswell there are three different types of ringbuffers: the render, the blitter, and the bitstream buffers.

Any application which uses the GPU is called a GPU *client*. Before submitting a job to the GPU, each client sends a *hardware-context creation request* to the Intel graphic driver. Hardware contexts are opaque objects which are used to manage context switches between clients. Intel added the support to hardware contexts from the Ironlake family (i.e. from 2010) relieving the driver from managing them. The clients submit a job *j* to the driver, which then tells the GPU to perform a context switch and execute *j*. To enforce the context switch, the driver encloses the commands received from the client with the proper *restore* and *save* commands. It is worth mentioning that a client can decide to avoid the creation of new hardware context, and in this case it gets associated with the so-called *default hardware context*. Nonetheless, in our experiments we focused on clients with an associated hardware context – as it is the case for Beignet.

The Intel graphic driver tracks all the hardware contexts using a linked list of i915_hw_context.

Memory management

From a memory forensics perspective it is very important to understand that the CPU and the *memory controller* have two different views of the memory. From the controller point of view, the memory is just an array of contiguous physical addresses. On the contrary, the CPU

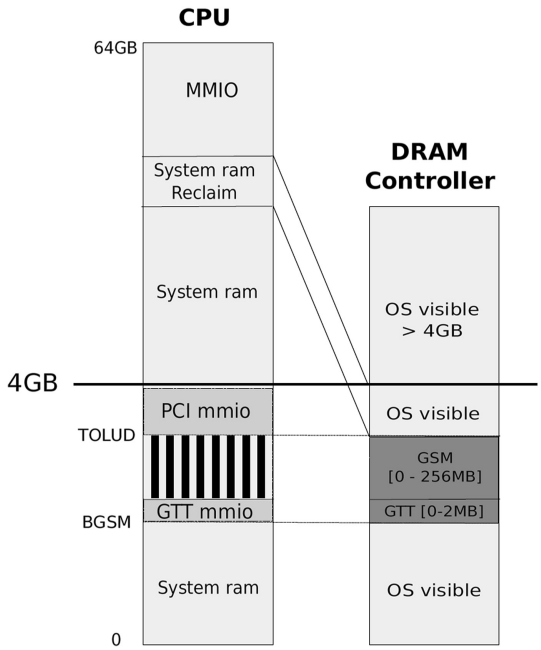


Fig. 3. The address space layout on Intel Haswell.

sees the memory as sequence of special-purpose areas of contiguous locations. Such areas include system RAM as well as MMIO. The combination of these special-purpose areas is normally called *Address Space Layout*. The address space layout of the Haswell architecture is represented in Fig. 3. The figure shows on the right side the view of the Haswell DRAM controller and on the left side the view of the CPU. The address space is divided into two parts (i.e., below and above the 4 GB threshold) and each part is composed by system RAM, MMIO areas, and remapped regions. The graphic card has a specially-reserved area of memory that is not accessible from the CPU. This section is called *Graphic Stolen Memory* (GSM) and consists of two contiguous ranges, which are configured by the BIOS: the *Graphics Translation Tables* (GTT) range that stores the virtual to physical graphic translation tables and the *Data Range* which is a programmable space, used to store graphics data (e.g the framebuffer).

A special register called *Base of GTT Stolen Memory* (BGSM in the Figure) points to the base address of the GSM. The TOLUD register marks instead the top of the low usable physical memory that is accessible to the CPU. The only way for the CPU to read the GTT memory is through MMIO on the PCI BAR0 of the graphic card. A MMIO on the PCI BAR2 — which is also called the *Graphics Aperture* — is used to access the Data range. The size of this area is defined in the BIOS and goes from 32 MB to 256 MB. The layout of the PCI BARs is depicted in Fig. 4.

IGDs have two virtual address spaces of 2 GB each, each one with its own page table. The first page table is called Global Graphic Translation Table (GGTT) and the second one is called Per-Process Graphic Translation Table (PPGTT). The GGTT is used by the render ring and the display functions. Interestingly, the GGTT entries can point both to the

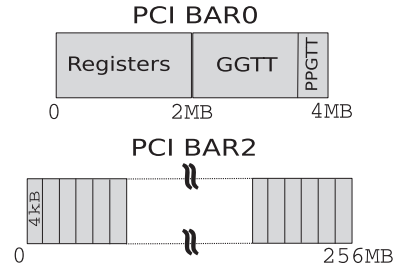


Fig. 4. The PCI bars for GPU MMIO.

GSM data range and to system RAM, while the PPGTT entries can point only to the system RAM. The two tables contain almost the same entries, except for those pointing to the data GSM that are only present in the GGTT.

The GGTT is a memory-resident page table containing an array of 32-bit Page Translation Entries (PTEs). The PPGTT is instead a two-level page table. The second level makes the allocation of memory for these structures less problematic for the operating system since it can be allocated also on not-contiguous pages. If a PPGTT is enabled, all rendering operations target per-process virtual memory.

Malware on Intel GPUs

In this section we describe the experiments we conducted to test how the different techniques presented in Section “GPU assisted malware” could be implemented on Intel GPUs.

Unlimited code execution

To obtaining unlimited code execution, a malware needs to disable the *hangcheck* in the Intel graphic driver. The hangcheck is a watchdog mechanism that kills any *kernel* running for more then *t* seconds. The i915 driver exposes a facility to disable the hangheck through the sysfs, at the path/sys/module/i915/parameters/enable_hangcheck (writing a zero to this file disables the watchdog). This operation requires superuser privileges.

Inconsistent memory mapping

When a buffer object is allocated through the GEM subsystem, it can be referenced from both the CPU and the GPU domains using two different virtual addresses. To implement this feature, the graphic driver creates two memory mappings in both the domains. However, the mapping in the CPU domain is stored into the OS page table whereas the mapping in the IGD domain is stored in the graphic page tables. During normal operations the two information are consistent. Therefore, as long as the corresponding physical address is not in the GSM, an analyst can extract the memory accessed by the GPU by looking at data structures of the operating system, in system memory.

However, if an entry in any of the graphic page tables is manually modified to point to a different location, the Linux page table is not affected. Indeed, the driver only cares of preserving the coherency between the GGTT and

the PPGTT. In this case, the virtual address in the CPU domain continues to point to the original physical page, while the GPU address now points to any arbitrary page in memory.

To show the feasibility of this attack we performed a simple test. Our experiment includes a victim application *V* which contains some sensitive data in a memory area B_{victim}. This application *does not* use the video card. At the same time, a malicious application *M* invokes the OpenCL library function clCreateBuffer() to allocate a buffer B_{dummy} of 1024 bytes. The malware then loads a kernel module that performs a number of tasks. First, it finds the physical address of B_{victim} and B_{dummy} and locates the Intel GPU in the PCI device list. Second, it retrieves the pointer to the MMIO address of the beginning of the Graphic Stolen Memory through the struct drm_i915_private data structure. Then, the module scans the two levels of the PPGTT to find the entry pointing to the physical address of B_{dummy} and modifies the entry to point to the physical address of B_{victim}. At this point, the malware unloads and removes the kernel module from the system. Finally, *M* prepares a *kernel* task *K* to overwrite the content of B_{dummy} (that in the IGD domains points to B_{victim}), it compiles it using the Beignet JIT compiler, and sends it to the video card.

Using this technique the malware forced an inconsistent memory mapping between the Linux page table and the GPU page table. An investigator looking at the system memory would see that the process *M* is using the GPU and that his *kernel* task has access to a buffer, which still correctly points to the harmless B_{dummy}. However, the GPU page table (which does not reside in the system RAM) allow the malware to overwrite the content of B_{victim}, which resides in a different process which does not even use the GPU. Even more interesting, we verified that the GPU is able to overwrite the target buffer even if it resides in a read-only page (for instance, we were able to modify the .text segment of another running process).

Process-less execution

We demonstrate that it is possible to have a *kernel* running in the GPU without any corresponding active host process. In this experiment the malware *M* uses a simple *kernel* that runs an infinite loop to increments the value of a properly allocated B_{dummy}. Using again the inconsistent memory mapping technique, the malware uses its *kernel K* to overwrite an integer variable which resides on the stack of a victim process.

However, in this experiment once the *kernel* execution started, we killed the malicious process by sending a SIGINT to *M* just after the *kernel* submission. In order to verify if the IGD was still running our code, we used several indicators. First, even if there was no trace of *M* in the process list, the value of the stack variable in the victim process kept increasing according to the code of *K*. Second, we used an utility to estimate the load of the render ring measuring the distance between the head and the tail pointers. Indeed, when the GPU is idle, the head and the tail pointers point to the same address. This was not the case in our experiment, confirming that something was indeed running in the GPU.

This attack only makes sense if the malware also disable the watchdog to perform an infinite task in the GPU. In a normal desktop computer, this would render the display not responsive and the attack could be easily noticed. However, this side effect is only noticeable when the IGD is effectively used for rendering. This is not the case of servers that are managed through remote connections or high end laptop and workstations which are equipped with both an integrated and the discrete GPUs. In such configurations is very common that the discrete GPU is the one used by default.

Context-less execution

We demonstrate that it is possible to have a *kernel* running in the GPU without any hardware context registered in the data structures of the Intel graphic driver. Indeed, the process-less execution still leaves some traces inside the Intel graphic driver. We were not able to retrieve the original malware PID, however, the buffer objects related to the running *kernel* and its hardware context, were still in the driver's memory.

Even though there is no a straightforward way to link such information back to the owner PID since the task struct was unlinked by the kernel after the SIGINT, this anomaly can be detected by a very careful forensic investigator.

Therefore, in our final experiment we tried to replicate the previous attack, but this time also to remove any hardware context associated with the running *kernel*. This technique, on top of requiring superuser privileges, also requires a detailed knowledge of the driver internals to perform a DKOM attack. In our test, we developed another malicious kernel module that locates the hardware context of the malware *kernel* and removes it from the internal list in the driver. To do so, the module performs the following steps: first, it looks for symbol referencing the struct drm_i915_private. Then it gets the context_list pointer and destroys the target context through the driver function i915_gem_context_unreference (). This operation has the side effect of also freeing all the buffer objects of the hardware context. However, as long as the malicious process is the only process running on the GPU, this operation does not alter the behavior of the malicious *kernel*. Finally, the malware set the memory of the hardware context to zero.

Findings and discussion

Table 1 summarizes the main findings of our study. The first column shows the different techniques we proposed in this paper and that can be used by malware to increase the complexity of a memory forensic examination. The second column shows the requirements of such malware: U for user privileges, S for super-user privileges, and K for super-user privileges with an additional knowledge of the GPU driver internal data structures. On the right side of the table we summarize the consequences for memory forensics, grouped around the three main objective of listing the processes that use the GPU, understanding which *kernel*

Table 1
Summary of our Findings.

Anti-forensic Technique	Malware Requirements	Forensic objectives		
		List processes	List kernels	Memory map
None	U	✓ (OS)	✓ (Driver)	✓ (OS)
Unlimited Execution	S	✓ (OS)	✓ (Driver)	✓ (OS)
Process-less	S	✗	✓ (Driver)	✓ (Driver)
Inconsistent Map	K	✓ (OS)	✓ (Driver)	✗
Context-less	K	✗	✗	✗

code is executed by each of them, and listing the memory ranges that are accessible to such *kernels*.

Memory forensic involves two separate processes: memory acquisition and memory analysis. Table 1 reports information for both processes. First, a ✓ sign means that the corresponding objective can be achieved by looking only at the content of the system memory. This is very important because it means that the memory acquisition tools we use today are already sufficient to complete the task.

When this is possible, we report if the memory analysis can be performed extracting information only from OS data structures, or if it requires ad-hoc modules to analyze the video driver internals. The latter can be quite problematic for a forensic point of view. Video drivers are often closed-source, and developing a separate memory analysis module (e.g., a Volatility plugin) for each of them can be a daunting task. A similar consideration applies to the cells containing the ✗ sign. In this case, an analyst would first need a specialized tool to dump the video card memory. In our scenario this means the Graphic Stolen Memory and the PCI BARs of the GPU, but this could be even more complicated for cards with a separate dedicated memory. Moreover, the analyst would then need custom modules to parse the data this memory contains. Again, without the support of the vendors, this can be completely impractical on a large scale.

Summary and guidelines

As part of our experiments, we developed a set of custom tools and volatility plugins to retrieve a number of information from the system and GPU memory. In particular, from a forensic perspective, we collected and parsed the following data structures:

Graphic page tables

This artifact can be used to detect the presence of inconsistent memory map in the system. For IGDs, the Graphic Translation Table (GTT) and Per-Process GTT (PGTT) can be dumped through mmio on the GPU PCI BAR2.

Hangcheck flag

This flag is stored in the driver and, as described in the previous section, can be very important to detect sophisticated attacks.

List of buffer objects

Together with the graphic page tables, these artifacts can be used to detect which memory pages can be accessed by a process, passing through the GPU. In the case of GEM, this information consists of two doubly-linked lists that can be obtained from the struct drm_i915_private.

List of contexts

This artifact can be used to get information on the GPU *kernels* and understand which processes are using the GPU. In IGDs this can be done through the context_list field in the struct drm_i915_private data structure.

Register file

It contains information on the internal state of the GPU and can be accessed through mmio on GPU PCI BAR0. Among the IO registers the RING_TAIL and the RING_HEAD deserve a special mention. These registers are used to control how the ringbuffer is accessed by the CPU and can be very useful for the analyst to understand if there were any tasks running during the acquisition process.

Beyond our case study

The variety of GPU ecosystem poses a big challenge for the forensic community. In the worst case, a different tools should be developed for each possible combination of GPU model and Operating system. Nonetheless, for what concerns Linux, the presence of the DRM layer can ease this task. Other than the Intel driver, also other drivers (e.g. tegra, nouveau, radeon, …) are compliant with both the DRM and the GEM memory manager. For such drivers, most of our considerations remains the same. For other closed-source drivers, more work is required to reverse-engineer their internal data structures.

Few additional observations can be done by looking at Table 1. For example, the information stored in the system memory (marked with the ✓ sign in the table) are likely to be the same for most GPUs. In fact, to use the GPU, a host process needs to communicate with the driver and in order to do this, the host process must open the GPU device file. Therefore, to find which host processes are using the GPU, the system memory is enough. Contrarily, especially for cards with dedicated memory, some of the information that in our case were stored within the driver's memory may instead reside in the device memory. This would translate inevitably into more ✗ locations on the table.

Related work

Several examples of GPU-assisted malware have been analyzed so far (Vasiliadis et al., Oct 2010; Ladakis et al., 2013; Triulzi, 2008; Danisevskis et al., 2014). For instance, in Ladakis et al. (2013) the GPU is used to implement a stealthy keylogger. To access the keyboard buffer, the GPU uses the physical addressing through DMA, without any CPU intervention. In (Vasiliadis et al., Oct 2010) the authors present a proof-of-concept which leverages GPU to unpack the code of a malware with a XOR-based encryption scheme using several random keys. To hinder the analysis, the malware's code is unpacked at runtime and the

decryption keys are stored in the device memory, that is not accessible from the CPU. In Triulzi (2008) the author shows how the GPU and NIC with a malicious firmware can be used to exfiltrate data from the host's memory. Finally, in Danisevskis et al. (2014) the authors describe an attack targeting a mobile GPU. In such case a vulnerability in the driver is exploited to take over the memory protection mechanism of the GPU.

Vasiliadis et al. (Vasiliadis et al., 2014) implemented some encryption algorithms with CUDA and leveraged the non-preemptive nature of GPUs to preserve the integrity of the cryptographic algorithms and the confidentiality of the key. However, the same technique can be used by malware to hinder live analysis or to achieve persistence.

In the past, several anti-memory forensic techniques have been developed. Such techniques can be divided into two categories: *anti-acquisition* and *anti-analysis* (Stttgen and Cohen, 2013). Anti-acquisition operates during the memory acquisition process interfering with the memory scanner. For instance, in Sparks and Butler (2005a) the author shows how a kernel rootkit can hijack the memory scanner reads to hide its presence in the memory dump. Some PoC tools are more invasive and they prevent the kernel from loading additional drivers (Memoryze, 2015). The goal of anti-analysis techniques is instead to prevent the correct analysis of the memory dump. They perform DKOM to prevent the memory analysis tools from finding some fundamental kernel variables which are used as starting point for the analysis (e.g. the _KDDEBUGGER_DATA64 data structure). Haruyama et al. (Haruyama & Suzuki, 2012) show how the modification of one-byte on some kernel variables can break the memory analysis process. Stuttgen et al. (Stttgen & Cohen, 2013) show that hooking memory enumeration and memory mapping API can hinder the analysis. This is substantially different from the Incosistent Mapping presented here. Indeed, even in the presence of the improved memory acquisition technique proposed in Stttgen and Cohen. (2013), the GPU page table would be marked as not safe to dump and the modified mapping would not be detected.

GPUs can be used as both an *anti-acquisition* and *anti-analysis* technique. Indeed, the non-preemptive nature of GPUs can be leveraged to prevent the memory acquisition process of the GPU memory itself. Pixelvault (Vasiliadis et al., 2014) locks the GPU to protect the key material, the same mechanisms can be used to prevent the memory scanner from getting the dump of the GPU memory. Furthermore, since GPUs are not considered during the memory-forensic analysis by current tools, using the GPU to run malicious code can prevent the correct analysis of the memory dump.

Conclusions and future work

In this work we analyzed to what extent a GPU can be used to perform malicious computation. We modeled the GPU as an anti-forensic tools and we highlighted four different techniques that a malware can use to hide its presence. We provided a case study on Intel Integrated GPUs showing how each of the four techniques impacts on the memory analysis. We tried to keep our analysis as general as possible however, as a future work, we plan to perform additional experiments targeting different GPU models and different Operating Systems.

References

ArsTechnica. Bitcoin malware uses gpu for mining. 2015. http://goo.gl/jdGHZV.

Beignet, 2015. https://01.org/beignet.

Cui A, Costello M, Stolfo SJ. When firmware modifications attack: a case study of embedded exploitation. In: NDSS; 2013.

Danisevskis J, Piekarska M, Seifert J-P. Dark side of the shader: mobile gpu-aided malware delivery. In: Information Security and Cryptology (ICISC). Springer International Publishing; 2014.

Di Pietro R, Lombardi F, Villani A. Cuda leaks: information leakage in gpu architectures. arXiv preprint. 2013., arXiv:1305.7383.

Freedesktop. The direct rendering infrastructure. 2015. http://dri.freedesktop.org/wiki/.

Haruyama T, Suzuki H. One-byte modification for breaking memory forensic analysis. In: BlackHat Europe Conference; 2012.

Intel, 2015. https://01.org/linuxgraphics/.

Kotcher R, Pei Y, Jumde P, Jackson C. Cross-origin pixel stealing: timing attacks using css filters. In: Proc. of the ACM SIGSAC Conference on Computer and communications security; 2013.

Ladakis E, Koromilas L, Vasiliadis G, Polychronakis M, Ioannidis S. You can type, but you can't hide: a stealthy gpu-based keylogger. In: Proc. of the 6th European Workshop on System Security (EuroSec); 2013.

Lee S, Kim Y, Kim J, Kim J. Stealing webpages rendered on your browser by exploiting gpu vulnerabilities. In: Proc. of the IEEE Symposium on Security and Privacy; 2014. Washington, DC, USA.

LiMe, 2015. http://goo.gl/IrKQ3P.

Maurice C, Neumann C, Heen O, Francillon A. Confidentiality issues on a gpu in a virtualized environment. In: Proc. of the 18th International Conference on Financial Cryptography and Data Security; 2014.

Memoryze. Meterpreter anti memory forensics script. 2015. http://goo.gl/lJS3a6.

Rekall. Memory dump with rekall. 2015. http://goo.gl/0DU5Bl.

Richard GG. Gpu malware research and the 2014 dfrws forensic challenge. 2015. http://goo.gl/Ly3ILv.

Sparks S, Butler J. Shadow walker: raising the bar for rootkit detection. In: BlackHat Japan; 2005.

Sparks S, Butler J. Raising the bar for windows rootkit detection. PHRACK, Issue 63. 2005.

Sparks S, Embleton S, Zou CC. A chipset level network backdoor: bypassing host-based firewall & ids. In: Proc. of 4th International Symposium on Information, Computer, and Communications Security. New York, NY, USA: ACM; 2009.

Stttgen J, Cohen M. Anti-forensic resilient memory acquisition. Digit Investig 2013;10(0). the Proc. of the 13th Annual Digital Forensics Research Conference (DFRWS).

Triulzi A. Project maux mk.ii. 2008. http://goo.gl/N4KdYR.

Vasiliadis G, Elias A, Polychronakis M, Ioannidis S. Pixelvault: using gpus for securing cryptographic operations. In: Proc. of the 21st ACM Conference on Computer and Communications Security. ACM; 2014.

Vasiliadis G, Polychronakis M, Ioannidis S. Gpu-assisted malware. In: Malicious and Unwanted Software (MALWARE), 2010 5th International Conference; Oct 2010. p. 1–6.

Vetter, D., 2015. http://goo.gl/rIU0bn.

Zaddach J, Kurmus A, Balzarotti D, Blass EO, Francillon A, Goodspeed T, et al. Implementation and implications of a stealth hard-drive backdoor. In: ACSAC, 29th Annual Computer Security Applications Conference; 2013. New Orleans, Louisiana, USA.