Noname manuscript No.
(will be inserted by the editor)

GPU-Assisted Malware

Giorgos Vasiliadis · Michalis Polychronakis · Sotiris Ioannidis

the date of receipt and acceptance should be inserted later

Abstract Malware writers constantly seek new methods to increase the infection lifetime of their malicious software. To that end, techniques such as code unpacking and polymorphism have become the norm for hindering automated or manual malware analysis and evading virus scanners. In this paper, we demonstrate how malware can take advantage of the ubiquitous and powerful graphics processing unit (GPU) to increase its robustness against analysis and detection. We present the design and implementation of brute-force unpacking and runtime polymorphism, two code armoring techniques based on the general purpose computing capabilities of modern graphics processors. By running part of the malicious code on a different processor architecture with ample computational power, these techniques pose significant challenges to existing malware detection and analysis systems, which are tailored to the analysis of CPU code. We also discuss how upcoming GPU features can be used to build even more robust and evasive malware, as well as directions for potential defenses against GPU-assisted malware.

Keywords GPU · malware · evasion

1 Introduction

Computer viruses, bot clients, rootkits, and other types of malicious software, collectively referred to as *malware*, abuse infected hosts to carry out their malicious activities. From the first viruses written directly in assembly language, to application-specific malicious code written in high-level languages such as JavaScript, any action of the malware results in the execution of machine code on the compromised system's processor.

Besides the central processing unit, personal computers are equipped with another powerful computational device: the graphics processing unit (GPU). Historically, the GPU has been used for handling 2D and 3D graphics rendering, effectively offloading these computationally-intensive operations from the CPU. Driven to a large extent by the ever-growing video game industry, graphics processors have been constantly evolving, increasing both in computational power and in the range of supported operations and functionality.

The most recent development in this evolving area is general-purpose computing on GPUs (GPGPU), which allows programmers to exploit the massive number of transistors in modern GPUs for performing computations that are traditionally handled by the CPU. In fact, leading vendors like AMD and NVIDIA have released software development frameworks that allow programmers to use a C-like programming language to write general-purpose code for execution on the GPU [8,28]. GPGPU has been used in a wide range of applications, while the increasing programmability and functionality of the latest GPU generations allows the code running on the GPU to fully cooperate with the host's CPU and memory.

Given the great potential of general-purpose computing on graphics processors, it is only natural to expect that malware authors will attempt to tap the powerful features of modern GPUs to their benefit [23, 24, 30]. Two key factors that affect the lifetime and potency of sophisticated malware are its ability to evade existing anti-malware defenses, and the effort required by a malware analyst to analyze and uncover its functionality—often a prerequisite for implement-

Giorgos Vasiliadis
FORTH, Greece
E-mail: gvasil@ics.forth.gr

Michalis Polychronakis
Columbia University, USA
E-mail: mikepo@cs.columbia.edu

Sotiris Ioannidis
FORTH, Greece
E-mail: sotiris@ics.forth.gr

ing the corresponding detection and containment mechanisms. To that end, packing and polymorphism are among the most widely used techniques for evading malware scanners [36]. Code obfuscation and anti-debugging tricks are commonly used to hinder reverse engineering and analysis of (malicious) code [14].

So far, these evasion and anti-debugging techniques take advantage of the intricacies of the most common code execution environments. Consequently, malware defense and analysis mechanisms, as well as the expertise of security researchers, focus on the most prevalent instruction set architectures (ISA), such as x86 and ARM. The ability to execute general purpose code on the GPU opens a whole new window of opportunity for malware authors to significantly raise the bar against existing defenses. The reason for this is that existing malicious code analysis systems primarily support x86 code, and the majority of security researchers are not familiar with the execution environment and ISA of graphics processors.

Our previous research [37] has demonstrated the feasibility of implementing malware that uses the GPU to armor its code. Specifically, we presented the design and implementation of GPU-based unpacking and runtime polymorphism, two techniques that pose significant challenges to existing malware detection and analysis systems.

In this paper, we extend our previous work [37] by showing how malware can tap the computational power of modern graphics processors to armor itself using *brute-force unpacking*, which can be performed more than an order of magnitude faster compared to a CPU implementation. We also discuss potential attacks and future threats that can be facilitated by next-generation GPGPU architectures, and discuss potential defense mechanisms, including both malware analysis and run-time detection techniques.

We believe that a better understanding of the offensive capabilities of attackers, as presented in this work, can lead researchers to create more effective and resilient defenses.

2 GPGPU Programming

General-purpose computing on graphics processing units has drastically evolved in recent years. As graphics processors started becoming more powerful, programmers began exploring ways for enabling their applications to take advantage of the massively parallel architecture of modern GPUs.

Standard graphics APIs, such as OpenGL and DirectX, do not expose much of the underlying computational capabilities that graphics hardware can provide. The task of using these APIs for general-purpose computation poses challenges when non-graphics applications are attempted to be ported to the GPU. Data and variables have to be mapped to graphics objects, while algorithms must be expressed as pixel or vertex shaders, pretending to perform graphics transformations. The lack of convenient data types, basic computational functionality, and a generic memory access model renders this environment far from attractive for developers accustomed to working in traditional programming environments.

The Compute Unified Device Architecture (CUDA) introduced by NVIDIA [28] is a significant advance, exposing several hardware features that are not available via the graphics API.[1] CUDA consists of a minimal set of extensions to the C language and a runtime library that provides functions to control the GPU from the host, as well as device-specific functions and data types.

At the top level, an application written for CUDA consists of a serial program running on the CPU, and a parallel part, called a *kernel*, that runs on the GPU. A kernel, however, can only be invoked by a parent process running on the CPU. As a consequence, a kernel cannot be initiated as a stand-alone application, and it strongly depends on the process that invokes it.

Each kernel is executed on the device as many different *threads* organized in thread *blocks*. The thread blocks are executed by the *multiprocessors* of the GPU in parallel. Each multiprocessor consists of eight *stream processors*, operating on a SIMT (Single Instruction, Multiple Thread) fashion. In order to maximize the use of the multiprocessors' computational resources, a thread scheduler periodically switches from one thread block to another.

In addition to program execution, CUDA also provides functions for data exchange between the host and the device. All I/O transactions are performed over the PCI Express bus. Furthermore, memory operations can be performed through DMA in order to facilitate concurrent execution between the CPU and the GPU. A block of page-locked host memory can also be mapped into the address space of the GPU, enabling the program running on the CPU and the kernel executing on the GPU to directly access the same data.

From the perspective of the malware author, a GPU-assisted malware binary contains code destined to run on different processors, as shown in Figure 1. Upon execution, the malware loads the device-specific code on the GPU, allocates a memory area accessible by both the CPU and the GPU and initializes it with any shared data, and schedules the execution of the GPU code. Depending on the design, the flow of control can either switch back and forth between the CPU and the GPU, or separate tasks can run in parallel on both processors.

A major advantage for the malware author is that the malware can be statically linked with the CUDA library into a single stand-alone executable. Thus, the malware becomes completely self-contained, without the need to install any files on the infected system. Furthermore, the execution of

[1] AMD offers a similar SDK for its ATI line of GPUs [8].

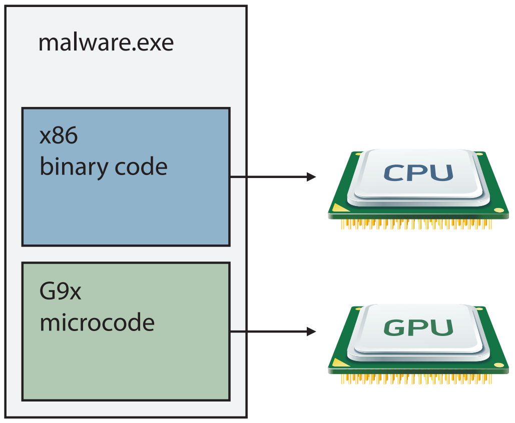


Fig. 1 A GPU-assisted malware binary.

GPU code, as well as data transfers between the host and the device do not require any administrator privileges. In other words, the malware will run successfully even under user privileges, making it more robust and deployable.

3 GPU-based Code Armoring

Malware can exploit the rich functionality of modern GPUs in a plethora of ways. In this section, we describe the design and implementation of code-armoring techniques based on GPU code. These prototype implementations not only demonstrate the feasibility of GPU-assisted malware, but also already pose significant challenges to existing malware analysis and detection systems.

We have chosen to implement our prototypes using the CUDA framework [28] by NVIDIA, probably the most widely used GPGPU framework. An attacker could easily include multiple versions of the GPU-specific code in the same executable to keep the malware functional across different GPU architectures. In fact, supporting just the two major vendors allows for covering a significant fraction of the PC market, as discussed in Section 4. The wide adoption of OpenCL [20], a cross-platform GPGPU framework that aims to unify vendor-specific APIs into a single one, will in the future obviate the need for embedding different versions of the same code for different architectures.

3.1 Basic Self-Unpacking

Code packing is one of the most common approaches malware writers employ to protect their code for evading detection [16]. Using this technique, the code of the malware is converted to data using compression, encryption, or any other data transformation technique. At runtime, an embedded decryption routine first *unpacks* the concealed code and then transfers control to the actual malicious code that has been revealed on the host's memory. Using variations in

```
1   .entry _Z8unpckrPhii (
2     .param .u32 __cudaparm__Z8unpckrPhii_a,
3     .param .s32 __cudaparm__Z8unpckrPhii_N,
4     .param .s32 __cudaparm__Z8unpckrPhii_key)
5   {
6     .reg .u32 %r<12>;
7     .reg .pred %p<4>;
8     .loc      28      31       0
9   $LBB1__Z8unpckrPhii:
10    ld.param.s32     %r1, [__cudaparm__Z8unpckrPhii_N];
11    mov.u32          %r2, 0;
12    setp.le.s32      %p1, %r1, %r2;
13    @%p1 bra         $Lt_0_1282;
14    ld.param.s32     %r1, [__cudaparm__Z8unpckrPhii_N];
15    mov.s32          %r3, %r1;
16    ld.param.u32     %r4, [__cudaparm__Z8unpckrPhii_a];
17    mov.s32          %r5, %r4;
18    add.u32          %r6, %r1, %r4;
19    ld.param.s32     %r7, [__cudaparm__Z8unpckrPhii_key];
20    mov.s32          %r8, %r3;
21  $Lt_0_1794:
22  //<loop> Loop body line 31, nesting depth: 1,
23  //estimated iterations: unknown
24    .loc      28      35       0
25    ld.global.u8     %r9, [%r5+0];
26    .loc      28      31       0
27    ld.param.s32     %r7, [__cudaparm__Z8unpckrPhii_key];
28    .loc      28      35       0
29    xor.b32          %r10, %r7, %r9;
30    st.global.u8     [%r5+0], %r10;
31    add.u32          %r5, %r5, 1;
32    setp.ne.s32      %p2, %r5, %r6;
33    @%p2 bra         $Lt_0_1794;
34  $Lt_0_1282:
35    .loc      28      37       0
36    exit;
37  $LDWend__Z8unpckrPhii:
38  } // _Z8unpckrPhii
```

Fig. 2 The intermediate PTX code of a simple XOR-based unpacking function for NVIDIA graphics cards. The main decryption loop iterates through the packed data (lines 24–33) one byte at a time. Each byte is XOR'ed with the specified key (lines 27–29).

the transformation method and the code of the decryption routine, as well as multiple layers of encryption, attackers are able to easily produce new variants of the same malware that can effectively evade existing detectors [31].

Implementing the self-unpacking functionality of a malware binary using GPU code can pose significant obstacles to current malware detection and analysis systems. Many systems for the automated extraction of packed executables inherently cannot handle GPU-based self-unpacking malware. For instance, PolyUnpack [31] relies on single-step execution and dynamic disassembly during the unpacking process. However, in contrast to x86 code, static and dynamic analysis of GPU machine code is at a nascent stage, and it is currently not supported by existing malware analysis systems.

Other unpacking systems like Renovo [19] monitor the execution of malware samples using a virtual machine. Unfortunately, existing virtual machine monitors provide only simulated graphics devices which currently do not support any GPGPU functionality. Thus, any malware sample that employs a GPU-based unpacking routine *will not run at all*

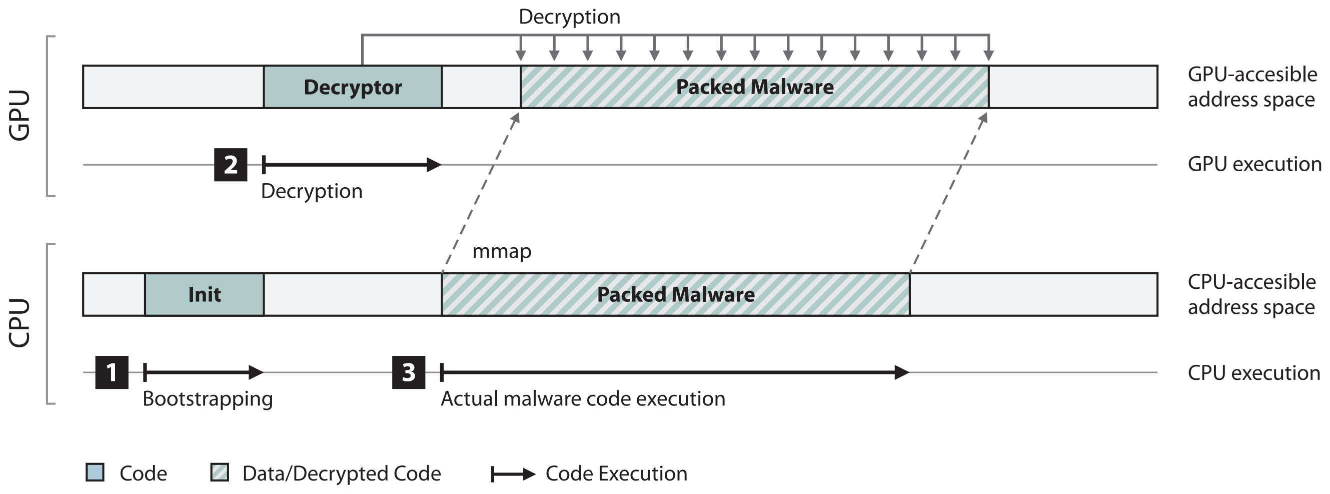


Fig. 3 Schematic representation of the execution of GPU-based self-unpacking malware. (1) During initialization, the malware loads the GPU code on the device, allocates a memory-mapped buffer accessible from both the CPU and the GPU, with the encrypted data, and transfers control to the GPU. (2) The decryption routine running on the GPU unpacks the encrypted code in the mapped buffer. (3) The unpacked code is executed on the CPU.

on a VM—a severe consequence for a multitude of existing dynamic malware analysis systems built on top of VMs and system emulators [9,22,27,33,38]. Although automated unpacking is still possible by systems that monitor for execute-after-write memory operations using only OS modifications [25], in practice, even these systems are usually deployed in combination with a virtual machine, for expediting system restore to a clean state and safeguarding malware execution.

For our basic proof-of-concept implementation of GPU-based unpacking we chose a simple XOR-based encryption scheme using a random key. The embedded unpacking function is compiled to device code of the underlying GPU instruction set. The intermediate code of the unpacking routine for CUDA, called PTX, is shown in Figure 2. Both the GPU unpacking function and the malware code are embedded into the same executable.

At start-up, the GPU code, usually referred to as the *kernel*, is loaded on the device and the CPU code starts executing, as shown in Figure 3. During the bootstrapping phase, the malware allocates a memory-mapped buffer that is used to store the packed binary data. As of CUDA v2.2 and later, it is possible to allocate and map an area of host memory that will be accessible from the device. Therefore, a kernel running on the GPU can access host memory directly, allowing the CPU and the GPU to share the same data. The flow of control is then transferred to the GPU, where the decryption routine unpacks the binary by modifying directly the mapped buffer. Upon decryption, control is transferred back to the CPU which executes the unpacked code.

The only CPU code that is exposed in the original malware image consists of the few instructions that copy the packed data to the newly allocated buffer and bootstrap the execution of the unpacking routine on the GPU. This minimal x86 code footprint does not leave much to existing static and dynamic malicious code analysis systems to actually analyze.

3.2 Brute-force Unpacking

Although even a simple GPU-based encoding scheme for malware packing can introduce significant hurdles for malware analysis, such an implementation exploits only the different nature of the graphics processor architecture, which is currently not supported by most analysis systems. However, a malware author can also take advantage of the computational power of modern graphics processors and pack the malware with extremely complex encryption schemes, which though can be efficiently computed due to the massively parallel architecture of GPUs. By relying heavily on GPU-friendly transformations—which are quite costly when implemented solely with CPU code—the same unpacking algorithm would take a prohibitively long time to complete when running solely on a CPU. This can severely affect existing malware scanners, which typically employ specific unpacking routines for different known packers to recover the original malware binary [25].

In our basic unpacking implementation, the decryption key is stored within the malware binary, and is used to unpack the encrypted malicious code. Regardless of the length and the complexity of the key being used, it is essential that the key is accessible by the malware in order to decode itself. Given the great potential of parallel processing in modern graphics cards, an alternative approach would be the following. The malware can simply be encrypted with a randomly generated key, which however is *not* included in the malware binary. At run-time, although the encryption key is not

Device	MD5	SHA1
CPU	28M trials/sec	20M trials/sec
GPU	609M trials/sec	404M trials/sec

Table 1 Brute-force unpacking rate for CPU and GPU implementations. Note that the CPU implementation runs in parallel on four CPU cores.

available, the malware can unpack its code using a brute-force scheme to guess the encryption key.

This approach is based on the fact that modern graphics processors have been shown to achieve great speedup on password guessing [3, 6]. In contrast to typical packed malware, which contains a decryption routine with an embedded decryption key, a malware that employs a brute-force unpacking scheme uses a brute-force key guessing engine to unpack the encrypted code.

To verify the correctness of the key, the malware could simply start executing the revealed instructions. However, this would result in the execution of incorrect code whenever the decryption was performed with a wrong key. In this case, the malware process would terminate, throwing an illegal instruction exception. A simpler scheme is to store a hashed version of the key along with the encrypted code. The brute-force engine can then try different key combinations, until its hash value matches the stored hashed key.

The strength of this approach is that the decryption key is not stored (in clear-text) in the malware itself, and thus cannot be extracted during malware analysis. At the same time, each instance of the same malware can easily be encrypted with a different randomly generated key. An important implication for conventional CPU-based automatic analysis systems is that they would need a much longer time to unpack the malware and reveal its functionality. For instance, custom packer-specific unpacking routines used in virus scanners, which currently run only on the CPU, will need a considerably larger amount of CPU time to decrypt a malware sample armored with a GPU-based brute-force unpacker. The difference in decryption speed may significantly delay the detection of the malware, and increase its propagation rate.

We implemented different versions of a brute-force unpacker using the MD5 and SHA1 hash functions. Both algorithms have been shown to achieve great performance in graphics processors [13, 21], while at the same time both have many optimized CPU implementations to compare with. After benchmarking of various implementations, we found the open source password cracker John The Ripper [4] to achieve the best performance among many others. John The Ripper uses highly optimized implementations of both algorithms that take advantage of specialized SSE instructions found in modern CPUs.

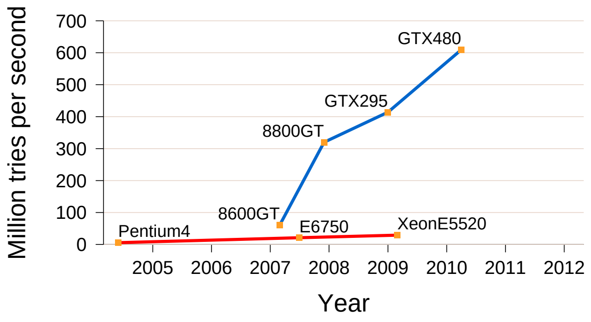


Fig. 4 Evolution of the brute force unpacking rate for various CPU and GPU models.

For our experiments, we used a system equipped with a quad-core Intel Xeon E5520 2.26 GHz with 8 MB cache. The graphics card was a NVIDIA GTX480, with 1.40 GHz clock rate and 1.5 GB of memory. Table 1 shows the key guessing rate achieved for both the CPU and GPU implementations. We can observe that the SHA1 GPU-based implementation is about 20 times faster than the corresponding CPU implementation, while MD5 is about 22 times faster. We should note that the CPU implementation takes advantage of all four cores of the system, as well as the optimized SSE instructions, to further parallelize the hashing operations.

A CPU-based unpacking routine for a malware that employs GPU-based brute force unpacking would require a considerably longer time to uncover the concealed malicious code. For example, using a key of length equal to six, the time required for decryption is about 20 minutes for the GPU, while the corresponding time for the CPU is more than seven hours.

To evaluate the difference in performance between the CPU and the GPU over time, we repeated the same experiment using CPU and GPU models of previous generations. Figure 4 shows how the performance gap between CPU and GPU models has been steadily growing over time. We observe that in less than two years, the computational throughput of GPUs has increased about 10 times, from 61.1M trials/sec to over 609M trials/sec. In contrast, the corresponding rate for CPUs has increased from 3M trials/sec at early 2007, to about 20M trials/sec in late 2010. We speculate that this trend will continue, and future GPU models will probably achieve even higher speedup.

3.3 Run-time Polymorphism

No matter how complex the encryption scheme in a packed malware is, upon the end of the unpacking process the code of the original malware will be restored on the host's memory. At that point, a malware analyst can take a snapshot

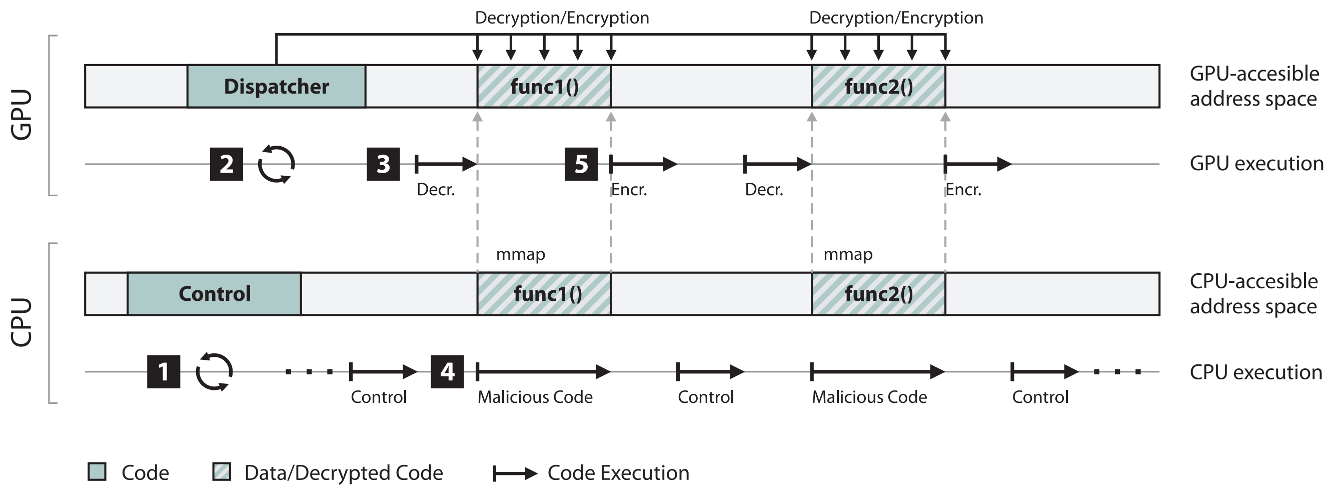


Fig. 5 Schematic representation of the execution of GPU-assisted malware that employs runtime polymorphism. (1) The CPU code of the malware carries out the main logic of the program by calling the appropriate function. (2) For each function to be executed, the dispatcher looks up its encrypted body and the corresponding decryption key and triggers its execution. (3) The decryption routine unpacks the encrypted code of the function into the mapped buffer. (4) The fuction is executed on the CPU. (5) The code of the function is re-encrypted using a different randomly generated key.

of the process' address space and analyze the exposed malicious code. Similarly, runtime malware scanners that inspect the address space of all running processes will be able to detect the original malicious code.

A well known technique to hinder the extraction of a complete process image is to decrypt only the parts of code needed at any given point in a just-in-time fashion [11]. Before decrypting a new part of code, any previously decrypted code that is no longer needed is re-encrypted. The finer the granularity of the on-demand decrypted parts, the smaller the code area that remains exposed on the host's memory.

For our proof-of-concept implementation of on-demand decryption based on GPU code, we chose *functions* as the unit of decryption. The machine code corresponding to each function in the original source code is encrypted separately using a different key. At run-time, the code of each function is decrypted on demand before its executed, and is re-encrypted just before returning to the caller.

Figure 5 shows how the GPU can be used for on-demand code decryption. All code for decrypting and re-encrypting each function resides entirely on the GPU, and thus the CPU is responsible for transferring control to the dispatching code running on the GPU right before and after function execution. Thus, during execution, the flow of control is constantly switching between the CPU and the GPU.

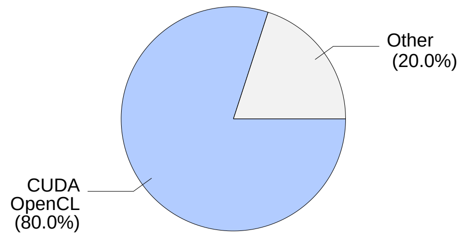
The encrypted code of each function is stored in memory segments that are accessible from both the CPU and the GPU. In contrast, the decryption keys are stored in private device memory that is *not* accessible from the CPU. This effectively precludes existing analysis methods that extract the keys and decrypt all encrypted code blocks using runtime instrumentation [12]. Moving a step further, after execution, each function is re-encrypted using a different randomly generated key, causing the malware to constantly mutate in unpredictable ways in the host's memory.

Although complete extraction of the original code is still possible by a determined malware analyst, when combined with existing anti-debugging techniques [12, 14], this form of GPU-assisted runtime polymorphism makes the whole reverse engineering process a challenging and time-consuming task. For instance, the GPU is a perfect fit for the implementation of runtime code checksumming, a quite effective anti-debugging technique [10]. In contrast to existing CPU-only implementations that use periodic checks, the GPU can constantly compute checksums of different code parts in a truly parallel fashion.

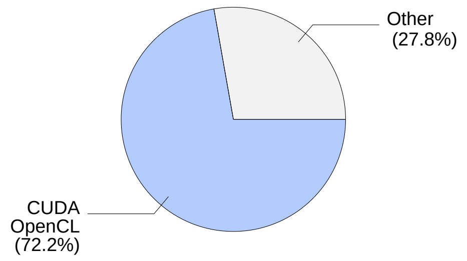
4 Infection Coverage

In this section, we explore the impact that GPU-assisted malware can have given the current technology landscape, by estimating the prevalence of graphics processors that support GPGPU. This is a crucial aspect for the feasibility of GPU-assisted malware, as it can be operational only on hosts equipped with graphics hardware that supports GPGPU.

Currently, the major vendors in the graphics processors industry are NVIDIA, AMD, and Intel, which collectively represent about 99% of the worldwide graphics cards market share [2]. Our proof-of-concept implementations of GPU-assisted malware, described in the previous sections, require a graphics card that supports CUDA or OpenCL. CUDA is specific to NVIDIA graphics processors, while OpenCL has been adopted by both AMD and NVIDIA. Intel also adopted OpenCL as of June 2012 for its 3rd Generation Intel Core Processors with Intel HD Graphics 4000/2500—



(a)



(b)

Fig. 6 Distribution of graphics processor models for the top-20 best-selling (a) laptop and (b) desktop personal computers (data source: Amazon [1]).

previous models can only be programmed using an existing graphics library, such as DirectX or OpenGL, even for general-purpose computing applications. As such, all new models manufactured by NVIDIA, AMD, and Intel can be programmed with OpenCL, and thus can easily be used by malware authors.

Figure 6 shows the percentage of graphics cards for the top-20 personal computers sold by a major online store [1], as of August 2014. We can see that 80% of the laptop and 72.2% of the desktop systems are equipped with a CUDA or OpenCL enabled GPU. The remaining systems are equipped with older generation Intel graphics processors, which do not support GPGPU.

It could be possible to implement some of the techniques described in section 3 using a graphics library, and extend the compatibility of a GPU-assisted malware to older-generation graphics processors. However, we do not consider this as an important obstacle for the impact of GPU-assisted malware, as it is reasonable to expect that all future models will include GPGPU functionality.

5 Defending Against GPU-Assisted Malware

Existing malware analysis and detection systems target malicious code only for CPU architectures, and are thus ineffective against GPU-assisted malware. Fortunately, GPU-assisted malware can be identified in several ways. To properly identify GPU-assisted malware though, existing defenses need to be enhanced with new functionality tailored to GPU code.

5.1 GPU Code Analysis

A first basic requirement is support for analyzing GPU machine code. NVIDIA has released `cuda-gdb` [28], a debugger for CUDA applications. The goal of `cuda-gdb` is to provide a mechanism of debugging in real time a CUDA application running on the actual GPU, similarly to `gdb(1)`. Unfortunately, `cuda-gdb` cannot execute programs that do not contain debug symbols (i.e., they have been compiled without the `-g` flag set on). Thus, it is not very useful for malware analysis, as an attacker can easily strip debug symbols from the malicious code.

An important requirement for systems built on top of virtual machine environments [9, 19, 22, 27, 33, 38] is the support of GPGPU APIs, in place of primitive graphics device emulation. Virtual machines usually provide a virtualization layer between the real graphics card of the host OS, and the emulated graphics card presented to the guest OS, allowing multiple VMs to access the same device. Therefore, when running on existing virtual machines, GPGPU applications fail to execute because the driver of the virtual graphics device does not support any of the GPGPU APIs.

Recently, NVIDIA released the SLI Multi-OS technology [5], which allows a user to assign a dedicated Quadro GPU to both the host operating system and a range of optionally loaded guest operating systems. The new technology creates a fully virtualized workstation, but requires a specific combination of software and hardware in order to get true access to the hardware. We believe though that this is a first step towards broader support of GPGPU APIs in virtualized environments.

In addition, gVirtuS [15] allows a virtual machine to run GPGPU programs in a transparent way, but with a greater overhead compared to a native GPGPU setup. Currently, gVirtuS supports only NVIDIA GPUs, however by design it is not limited to a particular GPU architecture and it is hypervisor-independent. Although the implementation of gVirtuS is still at an early stage, it could already be integrated in existing VM-based malware analysis systems.

5.2 GPU-Assisted Malware Detection

A possible mechanism for the detection of GPU-assisted malware can be based on the observation of DMA side effects. Stewin et al. [35] have shown that DMA malware has DMA side effects that can be reliably measured. However, the proposed technique works for DMA malware that performs bulk DMA transfers, e.g., continually searching the host's memory for valuable data to carry out an attack. However, since our GPU-assisted malware does not perform any such bulk transfers, it is not clear if this technique could be applied as an effective defense.

Alternatively, a possible defense can be based on monitoring the host memory of the GPU for code that is located within. Given that the host memory that is accessible from the GPU is allocated from a specific address range, a protection mechanism can monitor for any code residing or executing from that memory. However, an attacker can remap the GPU host memory to another address region. In current chipsets, the I/OMMU provides memory protection from misbehaving devices by taking care of mapping device addresses to physical addresses [7, 18]. However, it has been demonstrated that an I/OMMU configuration can be tricked with legacy PCIe devices [32]. Moreover, an I/OMMU can be attacked by modifying the number of DMA remapping engines provided by the BIOS [39]. This is done before the I/OMMU is configured by system software. Consequently, for a comprehensive protection against GPU-assisted malware, it is absolutely necessary to correctly configure the I/OMMU [34].

6 Discussion

In the previous sections we showed how malware can split its execution flow between the CPU and the GPU and thus evade existing anti-malware systems. While already powerful, the techniques described here utilize only a fraction of the functionality provided by modern GPUs. We expect that malware writers will soon start taking advantage of both the graphical and computational capabilities of graphics processors to a greater extent.

GPUs offer massive computational parallelism, which can be used to speed up various CPU-intensive operations that malware may need to carry out. For example, a botnet can be set up for large-scale password cracking—a task that GPGPUs excel in [6, 17]. Bots can easily be extended with GPGPU support and then use the GPUs of infected hosts to offload password cracking.

This would not only result in a significant increase in overall password cracking performance, but would also hide the ongoing malicious activity. Since the GPU workload cannot be monitored in real time to identify what code is running, it is quite difficult to determine the presence of the password cracking code on the GPU. Additionally, the CPU will not be occupied at all during this computationally-intensive process, and thus CPU-load monitors will not be helpful in detecting the malicious activity.

Now, note that the *framebuffer*, which is part of the device memory of the GPU, contains what is displayed on the monitor at any given time. Having unrestricted access to the framebuffer opens the way for a wide range of possible attacks. For instance, malicious code running on the GPU could access the screen buffer periodically and harvest private data displayed on the user screen, and do so in a more stealthy way than existing screen capture methods. As a matter of fact, it has been recently shown that a user program's data stored in GPU memory can be revealed both during its execution and right after its termination [24, 26, 29]. Alternatively, more sophisticated malware could attempt to trick users by displaying false, benign-looking information when visiting rogue web sites (e.g., overwriting suspicious URLs with benign-looking ones in the browser's address bar).

We are currently exploring the feasibility of such attacks using existing GPUs. The framebuffer memory in current GPGPU architectures is protected from read and write operations. However, as vendors constantly try to improve the graphics interoperability between GPGPU SDKs and graphics APIs such as OpenGL and DirectX, it is quite possible that in future releases a kernel will have full access to the screen framebuffer. Accessing the screen pixels directly will increase the performance of many graphics operations, such as 3D transformations and video compression and decompression, by reducing data transfers between the CPU and the GPU. So this is a feature that may be part of future hardware releases.

Going one step further, future GPGPU architectures may enable the implementation of GPU-hosted malware, i.e., malicious software that runs solely on the GPU, without any association with a process running on the CPU. However, a major restriction of current graphics hardware architectures is that they do not support multitasking; only one task can utilize the GPU at any time. Hence, if the malware is launched directly on the GPU, without the intervention of the CPU, it may parasitically consume all GPU cycles without being context-switched. Consequently, the task responsible for rendering the monitor would not run, and the display would freeze. Although this scenario seems quite unrealistic right now, as many technical hurdles need to be overcome, it is possible that future graphics hardware will have the necessary functionality for next-generation malicious code that fully utilizes the GPU.

7 Conclusion

The rapid evolution of general-purpose computing on graphics processors enables malware authors to take advantage of the GPU present in modern personal computers, and increase the robustness of their code against existing defenses. The GPU-based code armoring techniques presented in this paper—basic self-unpacking, brute force unpacking, and run-time polymorphism—not only demonstrate the feasibility of GPU-assisted malware, but also show the great potential that general-purpose computing on GPUs has in enhancing the evasiveness and functionality of malicious code. All the techniques presented in this paper have been implemented and tested using typical graphics hardware that is widely available in recent systems, and pose significant challenges to existing malware analysis and detection systems, which mostly handle only CPU code.

Taking a step further, we describe potential attacks that malware can employ using upcoming features of next-generation GPUs. The constantly enhanced capabilities of graphics processors, for both graphics and general-purpose computations, can make graphics cards a promising environment for future malware. Fortunately, existing defenses can be extended to support the analysis of GPU machine code, and there already have been some initial steps towards this direction.

Acknowledgments

This work was supported in part by the Marie Curie Actions – Reintegration Grants project PASS, by the Marie Curie FP7-PEOPLE-2009-IOF project MALCODE, by the project i-Code funded by the Prevention, Preparedness and Consequence Management of Terrorism and other Security-related Risks Programme of the European Commission – Directorate-General for Home Affairs, by the General Secretariat for Research and Technology in Greece with a Research Excellence grant, and by the FP7 projects NECOMA and SysSec, funded by the European Commission under Grant Agreements No. 608533 and No. 257007. This publication reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained herein. Giorgos Vasiliadis is also with the University of Crete.

References

1. Amazon.com: Online Shopping for Electronics, Apparel, Computers, Books, DVDs, & more. http://www.amazon.com.
2. AMD Loses Share on Graphics Market. http://www.xbitlabs.com/news/video/display/20101026180100_AMD_Loses_Share_on_Graphics_Market.html.
3. GPU-Accelerated Wi-Fi password cracking goes mainstream. http://www.zdnet.com/blog/security/gpu-accelerated-wi-fi-password-cracking-goes-mainstream/2419.
4. John the Ripper password cracker. http://www.openwall.com/john/.
5. NVIDIA SLI Multi-OS. http://www.nvidia.co.uk/object/sli_multi_os.html.
6. Russian crackers throw GPU power at passwords. http://arstechnica.com/business/news/2007/10/russian-crackers-throw-gpu-power-at-passwords.ars.
7. Advanced Micro Devices, Inc. AMD I/O Virtualization Technology (IOMMU) Specification License Agreement. http://support.amd.com/us/Processor_TechDocs/48882.pdf.
8. AMD. ATI Stream Software Development Kit (SDK) v2.1. http://developer.amd.com/gpu/ATIStreamSDK/Pages/default.aspx.
9. U. Bayer and F. Nentwich. Anubis: Analyzing Unknown Binaries, 2009. http://anubis.iseclab.org/.
10. P. Biondi and F. Desclaux. Silver Needle in the Skype. BlackHat Europe 2008.
11. J. Cappaert, B. Preneel, B. Anckaert, M. Madou, and K. D. Bosschere. Towards tamper resistant code encryption: Practice and experience. In *Proceedings of the 4th Information Security Practice and Experience Conference (ISPEC)*, 2008.
12. C. Eagle. Strike/Counter-Strike: Reverse Engineering Shiva. BlackHat Federal 2003.
13. Elcomsoft. Faster Password Recovery with modern GPUs. http://www.elcomsoft.com/presentations/faster_password_recovery_with_modern_GPUs.pdf.
14. P. Ferrie. Anti-Unpacker Tricks. In *Proceedings of the 2nd International CARO Workshop*, 2008.
15. G. Giunta, R. Montella, G. Agrillo, and G. Coviello. gVirtuS: A GPGPU transparent virtualization component. http://osl.uniparthenope.it/projects/gvirtus/.
16. grugq and scut. Armouring the ELF: Binary encryption on the UNIX platform. *Phrack*, 11(58), Dec. 2001.
17. O. Harrison and J. Waldron. Practical symmetric key cryptography on modern graphics hardware. In *Proceedings of the 17th USENIX Security Symposium*, 2008.
18. Intel Corporation. Intel Virtualization Technology for Directed I/O - Architecture Specification. http://download.intel.com/technology/computing/vptech/Intel(r)_VT_for_Direct_IO.pdf.
19. M. G. Kang, P. Poosankam, and H. Yin. Renovo: a hidden code extractor for packed executables. In *Proceedings of the 2007 ACM workshop on Recurring Malcode (WORM)*, 2007.
20. Khronos Group. OpenCL - The open standard for parallel programming of heterogeneous systems. http://www.khronos.org/opencl/.
21. L. Koromilas, G. Vasiliadis, I. Manousakis, and S. Ioannidis. Efficient Software Packet Processing on Heterogeneous and Asymmetric Hardware Architectures. In *Proceedings of the 10th ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, ANCS, 2014.
22. C. Kruegel, E. Kirda, and U. Bayer. TTAnalyze: A Tool for Analyzing Malware. In *Proceedings of the 15th European Institute for Computer Antivirus Research Annual Conference (EICAR)*, April 2006.
23. E. Ladakis, L. Koromilas, G. Vasiliadis, M. Polychronakis, and S. Ioannidis. You can type, but you can't hide: A stealthy GPU-based keylogger. *Proceedings of the 6th European Workshop on System Security (EuroSec)*, 2013.
24. S. Lee, Y. Kim, J. Kim, and J. Kim. Stealing Webpages Rendered on Your Browser by Exploiting GPU Vulnerabilities. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, 2014.
25. L. Martignoni, M. Christodorescu, and S. Jha. OmniUnpack: Fast, Generic, and Safe Unpacking of Malware. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC)*, 2007.
26. C. Maurice, C. Neumann, O. Heen, and A. Francillon. Confidentiality Issues on a GPU in a Virtualized Environment. In *Proceedings of the Eighteenth International Conference on Financial Cryptography and Data Security*, FC 14, March 2014.
27. A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *Proceedings of the 28th IEEE Symposium on Security and Privacy*, 2007.
28. NVIDIA. Compute Unified Device Architecture (CUDA) Toolkit, version 3.2. http://developer.nvidia.com/object/cuda_3_2_downloads.html.
29. R. D. Pietro, F. Lombardi, and A. Villani. CUDA Leaks: Information Leakage in GPU Architectures. *ArXiv*, May 2013.

30. D. Reynaud. GPU Powered Malware. Ruxcon 2008.
31. P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee. PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC)*, 2006.
32. F. L. Sang, E. Lacombe, V. Nicomette, and Y. Deswarte. Exploiting an I/OMMU vulnerability. In *Proceedings of the 5th International Conference on Malicious and Unwanted Software (MALWARE)*, 2010.
33. M. Sharif, A. Lanzi, J. Giffin, and W. Lee. Automatic reverse engineering of malware emulators. In *Proceedings of the 30th IEEE Symposium on Security and Privacy*, 2009.
34. P. Stewin and I. Bystrov. Understanding DMA Malware. In *Proceedings of the 9th Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, DIMVA2012, July 2012.
35. P. Stewin, J.-P. Seifert, and C. Mulliner. Poster: Towards detecting DMA malware. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 857–860, 2011.
36. P. Ször. *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, February 2005.
37. G. Vasiliadis, M. Polychronakis, and S. Ioannidis. GPU-Assisted Malware. In *Proceedings of the 5th International Conference on Malicious and Unwanted Software (MALWARE)*, 2010.
38. C. Willems, T. Holz, and F. Freiling. Toward automated dynamic malware analysis using CWSandbox. *IEEE Security and Privacy*, 5(2):32–39, 2007.
39. R. Wojtczuk, J. Rutkowska, and A. Tereshkin. Another Way to Circumvent Intel Trusted Execution Technology. `http://invisiblethingslab.com/resources/misc09/Another%20TXT%20Attack.pdf`, 2009.