You Can Type, but You Can't Hide: A Stealthy GPU-based Keylogger

ABSTRACT

Keyloggers are a prominent class of malware that harvests sensitive data by recording any typed in information. Keylogger implementations strive to hide their presence using rootkit-like techniques to evade detection by antivirus and other system protections. In this paper, we present a new approach for implementing a stealthy keylogger: we explore the possibility of leveraging the graphics card as an alternative environment for hosting the operation of a keylogger. The key idea behind our approach is to monitor the system's keyboard buffer directly from the GPU via DMA, without any hooks or modifications in the kernel's code and data structures. The evaluation of our prototype implementation shows that a GPU-based keylogger can effectively record all user keystrokes, store them in the memory space of the GPU, and even analyze the recorded data in-place, with negligible runtime overhead.

1. INTRODUCTION

Keyloggers are one of the most serious types of malware that surreptitiously log keyboard activity, and typically exfiltrate the recorded data to third parties [16]. Despite significant research and commercial efforts [21, 20, 26, 25], keyloggers still pose an important threat of stealing personal and financial information [1]. Keyloggers can be implemented as tiny hardware devices [3, 4], or more conveniently, in software [5].

Software keyloggers can be implemented either at the user or kernel level. User-level keyloggers generally use high-level APIs to monitor keystrokes. For example, Windows provides the `GetAsyncKeyState` function to determine whether a key is pressed or not at the time the function is called, and whether the key was pressed after a previous call. User-space keyloggers, while easy to write, are also relatively easy to detect, using hook-based techniques [32]. In contrast, kernel level keyloggers run inside the OS kernel and record all data originating from the keyboard. Typically, a kernel level keylogger hooks specific system calls or driver functions. The injected malicious code is programmed to capture all user keystrokes passed through the hooked function call. Although kernel-level keyloggers are more sophisticated and stealthy than user-level keyloggers, they heavily rely on kernel code modifications, and thus can be detected by kernel integrity and code attestation tools [30, 29].

In this paper, we present how malware can tap the general-purpose computation capability of modern graphics processors to increase the stealthiness of keylogging. By instructing the GPU to carefully monitor via DMA the physical page where the keyboard buffer resides, a GPU-based keylogger can effectively record all user keystrokes and store them in the memory space of the GPU.

An important property of our GPU-based keylogger is that it does not rely on any kernel modifications. It only uses a small code snippet that needs to run once from kernel context to acquire the physical address of the keyboard buffer. This code is completely standalone, does not require any hooks or other modifications, and is completely removed after it executes its task. The physical address of the keyboard buffer is then used by the GPU to monitor all user keystrokes directly via DMA.

We have implemented and evaluated our prototype GPU-based keylogger for Linux and the 32-bit x86 architecture. The results of our evaluation demonstrate the efficiency and effectiveness of GPUs for capturing short-lived data from the host memory, while keeping the CPU and GPU utilization at minimal levels, in our case about 0.1%.

The main contributions of this work are:

- We present the first (to the best of our knowledge) GPU keylogger, capable of monitoring all keystroke events and storing them in the GPU memory.
- We have experimentally evaluated our GPU-based keylogger and demonstrate that it can be used effectively and efficiently for capturing all keystroke events. By carefully scheduling the GPU kernel invocations, the keylogger can capture all keystroke events without affecting the proper rendering of the graphics and with minimal overhead.

2. BACKGROUND

General-purpose computing on graphics processing units has drastically evolved during the last few years. Historically, the GPU has been used for handling 2D and 3D graphics rendering, effectively offloading the CPU from these computationally intensive operations. 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. Meanwhile, 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 [7] and DirectX [6], 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. 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 [23] 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 CPU process that invokes it. Each kernel is executed on the device as many different *threads* organized in thread *blocks*. Thread blocks are executed by the *multiprocessors* of the GPU in parallel. Each multiprocessor consists of eight *stream processors*, operating in a SIMD fashion. 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 appropriate functions for data exchange between the host and the device. All I/O transactions are performed via DMA over the PCI Express bus. DMA enables the GPU to trasfer data directly—without any CPU involvement—to and from the host memory, using a dedicated DMA engine. Typically, the GPU can only access specific memory regions, allocated by the operating system.

Given the great potential of general-purpose computing on graphics processors, it is only natural to expect that malware authors would attempt to tap the powerful features of modern GPUs to their benefit [28, 36]. 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, while current virus scanning tools cannot detect malicious code stored in separate device memory and executed on a processor other than the CPU. In addition, the majority of security researchers are not familiar with the execution environment and the instruction set of graphics processors.

A GPU-assisted malware binary contains code destined to run on different processors. 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 malware authors is that the majority of current video card manufacturers, representing about 99% of the worldwide graphics cards market share [8], do provide support for GPGPU computations. Consequently, GPU-based malware can have a large infection ratio without being inhibited by unsupported graphics processors. In addition, the execution of GPU code and data transfers between the host and the device do not require any administrator privileges. In other words, depending on its purpose, GPU-assisted malware can run successfully even under user privileges, making it more robust and deployable.

3. GPU-BASED KEYLOGGING

In this section we present in detail the design of a proof-of-concept keylogger implemented on the GPU. Instead of relying on rootkit-like techniques, such as hooking system functions and manipulating critical data structures, our keylogger monitors the contents of the system's keyboard buffer directly from the GPU.

One of the primary challenges of this design is how to locate the memory address of the keyboard buffer, as *(i)* the keyboard buffer is not exported in the kernel's symbol table, making it not accessible directly by loadable modules, and *(ii)* the memory space allocated for data structures is different after every system boot or after unplugging and plugging back in the device. Typically, loadable modules allocate memory dynamically, hence object addresses are not necessarily the same after a system reboot. In addition, the OS can apply certain randomization algorithms to hinder an attacker that tries to predict an object's address.

To overcome the randomized placement of the keyboard buffer, the attacker has to scan the whole memory. As a consequence, our GPU-based keystroke logger consists of two main components: *(i)* a CPU-based component that is executed once, at the bootstrap phase, with the task of locating the address of the keyboard buffer in main memory, and *(ii)* a GPU-based component that monitors, via DMA, the keyboard buffer and records *all* keystroke events.

3.1 Locating the Keyboard Buffer

In Linux, an attached USB device is represented by a USB Request Block (URB) structure, defined in the `include/linux/usb.h` header file of the Linux source tree. Figure 1 shows the fields of the USB Request Block structure that are relevant for our work. For a USB keyboard device, in particular, the keyboard buffer is part of the URB structure, in the field `transfer_buffer`.

Unfortunately, the memory offset where the URB structure is placed is different every time the system restarts. To locate the exact offset of the keyboard buffer, we have to scan the whole memory sequentially [34]. However, modern OSes, including Linux and Windows, do not allow users to access memory regions that have not been assigned to them. An access to a page that is not mapped to a process' virtual address space is typically considered illegal, resulting in a segmentation fault. To access the memory regions where the OS kernel and data structures reside, the memory scan-

[1] AMD and Intel offer similars SDKs for its ATI line of GPUs [9] and the Intel HD Graphics 4000/2500 [2].

...
struct usb_device `*dev`
...
void `*transfer_buffer`
dma_addr_t `*transfer_dma`
...
u32 `*transfer_buffer_length`
...

Figure 1: Fields of interest in the USB Request Block (URB) structure.

ning phase of the keylogger needs to run with administrative privileges.

Linux offers the `/dev/mem` and `/dev/kmem` special files to allow a privileged user to access the physical memory and the kernel virtual memory, respectively. For security reasons though, recent distributions disable them by default; access to the `/dev/mem` and `/dev/kmem` files is allowed only if the Linux kernel has been explicitly compiled without the option `CONFIG_STRICT_DEVMEM=y`. Instead, we have implemented a loadable kernel module (LKM) that scans the whole main memory of the host. The kernel module uses the same mechanism as the `/dev/mem` character device to implement access to physical pages.

The pseudocode for scanning the low memory addresses of a 32-bit x86 system is shown in Figure 2. This approach is sufficient for memory allocated using `kmalloc()`, which always returns kernel virtual addresses that have a physical mapping (logical addresses) [12]. To locate the keyboard buffer, we begin to search for pointers to USB device structures. Such pointers are memory-aligned to 0x400 boundaries, and the corresponding `transfer_dma` fields are aligned to 0x20 boundaries. If both conditions are true, we check if the `product` field contains any of the substrings "usb" and "keyboard" (for wired USB keyboards), or "usb" and "receiver" (for wireless keyboard/mouse sets). As a final step, we check that the field `transfer_buffer_length` contains the appropriate length (8 bytes) and that it contains valid keystroke values, e.g., all bytes are zero if no key is pressed. For 32-bit systems, in which the kernel address space is at most 1 GB, the total search time in the worst case is just about 3.2 seconds.[2]

3.2 Capturing Keystrokes

Having located the memory address of the buffer used by the keyboard device driver, the next step is to configure the GPU to constantly monitor its contents for changes. To achieve this, the GPU must have access to the kernel's keyboard buffer. NVIDIA CUDA devices share a unified address space with the host *controller process* that manages the GPU [23]. Consequently, to be accessible directly by the GPU, the keyboard buffer must be mapped in the virtual address space of the host process. This can be achieved by manipulating the page table of the controller process to include the page in which the keyboard buffer resides.

During the initialization phase, the controller process acquires a dummy memory page using the `mmap()` system call. After the completion of the scanning phase, the bootstrapping kernel module locates the controller process' page table and changes the virtual mapping of the dummy page to

[2]In 64-bit architectures, where the kernel virtual address space can be larger than 1 GB, the search time grows linearly and is proportional to the size of the physical memory.

```
#define __va(x) ((void *)((unsigned long)(x)+PAGE_OFFSET))

for (i = 0; i < totalmem; i += 0x10) {
    struct urb *urbp = (struct urb *)__va(i);

    if (((urbp->dev % 0x400) == 0) &&
        ((urbp->transfer_dma % 0x20) == 0) &&
        (urbp->transfer_buffer_length == 8) &&
        (urbp->transfer_buffer != NULL) &&
        strncmp(urbp->dev->product, "usb", 32) &&
        strncmp(urbp->dev->product, "keyboard", 32)) {

        /* potential match */
    }
}
```

Figure 2: Pseudocode for locating the keyboard buffer. Whenever the condition of the `if`-statement is true, a potential URB structure of interest has been found. We verify whether a matching structure corresponds to the keyboard device by checking if the content of the `transfer_buffer` field conforms to the appropriate format, i.e., contains valid keystroke values.

point to the physical page that contains the keyboard buffer. After the GPU begins monitoring the buffer, the controller process immediately releases the page using `munmap()`. By doing so, the controller process does not account for that page any longer, allowing it to evade potential anomaly detection tools that check for suspicious page table mappings. Note that this does not affect the ability of the GPU to access the keyboard buffer, as it uses physical addressing through DMA, without any CPU intervention. In essence, the virtual mapping is only initially required to "trick" the CUDA API to allow DMA access to a physical page that otherwise would not be accessible.

To capture keystroke events, the GPU constantly monitors the buffer for changes. As we discuss in Section 4, an interval of less than 100 ms allows the recording of all keystrokes even for fast typists, with minimal runtime overhead and without adding any contention due to consecutive accesses. The size of the buffer is eight bytes.[3] The first byte corresponds to the scancodes of modifier keys, such as `Alt`, `Shift`, and `Ctrl`. If more than one modifiers are active at the same time, this value is encoded as the sum of the individual scancodes. The second byte has no special use. The last six bytes represent the scancodes of the rest of the pressed keys. At any given moment, the buffer may contain one to six non-zero bytes that represent the pressed keys. Whenever a user presses a key (or a combination of keys), the corresponding scan codes are written in the buffer, and remain there as long as the key(s) are pressed. By the time the user releases the key(s), the corresponding values are zeroed. An error state occurs when several keys are pressed simultaneously, and is represented by two zeroes followed by six ones.

Captured keystrokes are translated from raw scan codes into ASCII characters using a simple dispatcher, and are stored in the device memory of the GPU. Modern GPUs contain from many hundreds of MBs, up to 2–3 GBs of memory,

[3]In Linux, the `usbhid` keyboard driver allocates an 8-byte memory area for the `transfer_buffer` field of the USB Request Block that is used to handle the keyboard interrupts.

which is plenty for storing the recorded keystrokes. Furthermore, the parallel capabilities of modern GPUs can also be exploited to analyze the captured data, e.g., for extracting sensitive data such as credit card numbers and web-banking credentials. We have implemented a very simple module that performs regular expression matching—using an existing GPU-based pattern matching implementation [37]—over the recorded keystrokes periodically. As shown in Section 4, the GPU is capable of matching tens of MBs in less time than the time needed for a single user key press.

4. EVALUATION

To evaluate our prototype GPU-based keylogger, we used a commodity desktop equipped with an Intel E6750 Dual-Core CPU at 2.66GHz and 4GB of main memory. We use several NVIDIA graphics cards: both low-end (GT630) and high-end (GTX480). Our desktop runs Ubuntu Linux 12.10 with kernel v3.5.0. We measure GPU execution times using CUDA's command line profiler facilities [24].

In our first experiment, we measure the CPU and GPU utilization of the keylogger. The CPU time corresponds to the controller process, which periodically just makes a simple function call, provided by the GPU driver, instructing the GPU to invoke the keylogging GPU kernel function. The GPU kernel function reads the keyboard event buffer, occasionally performs simple data analysis tasks, and returns to the controller process, which remains idle for a specified interval. This approach is necessary because the GPU is also used for graphics rendering, and longer execution times of the GPU component would affect the proper display of graphics. More importantly, current GPUs use a non-preemptive scheduling mechanism, hence a running task cannot be interrupted.

This introduces an interesting trade-off: As the frequency of the GPU kernel function invocation increases, so does the CPU and GPU overhead of the keylogger, and the risk of affecting the proper display of graphics—an event which the user may notice. On the other hand, less frequent kernel invocations do not have any noticeable impact in graphics rendering, but may result to missed keystroke events. Figure 3 shows the keylogger's CPU and GPU utilization when varying the GPU kernel invocation interval. Typically, the duration of a single keypress varies from 100 ms for faster typists, to over one second for slower typists [18]. Consequently, the GPU invocation interval should be kept under 100 ms, to enable accurate monitoring of all key presses, without missed events. As shown in both figures, we have chosen an interval of 90 ms, which has minimal performance impact: the CPU utilization is about 0.1% (Figure 3(a)), while the GPU has negligible utilization of $5 \cdot 10^{-5}$ % (Figure 3(b)). The time needed by the GPU to read the contents of the 8-byte keyboard buffer over the PCIe bus is about 0.005 ms.

In the next experiment, we measure the time needed by the GPU to scan the captured data and extract sensitive information. Specifically, we search the recorded data for various types of credit card numbers, using the regular expressions shown in Table 1. Figure 4 shows the corresponding GPU execution times for different input sizes. We observe that the running times are below one millisecond even for buffer sizes in the order of Megabytes. As such, the scanning overhead is negligible, given that the average user needs several seconds to type a few hundred of bytes. Data analysis can be performed infrequently, e.g., after the accumulation of a few Megabytes of new data.

Type	**Regular Expression**
VISA	$^\wedge 4[0-9]\{12\}(?:[0-9]\{3\})?\$$
MasterCard	$^\wedge 5[1-5][0-9]\{14\}\$$
American Express	$^\wedge 3[47][0-9]\{13\}\$$
Diners Club	$^\wedge 3(?:0[0-5]\vert[68][0-9])[0-9]\{11\}\$$
Discover	$^\wedge 6(?:011\vert 5[0-9]\{2\})[0-9]\{12\}\$$

Table 1: Regular expressions used for matching various types of credit card numbers.

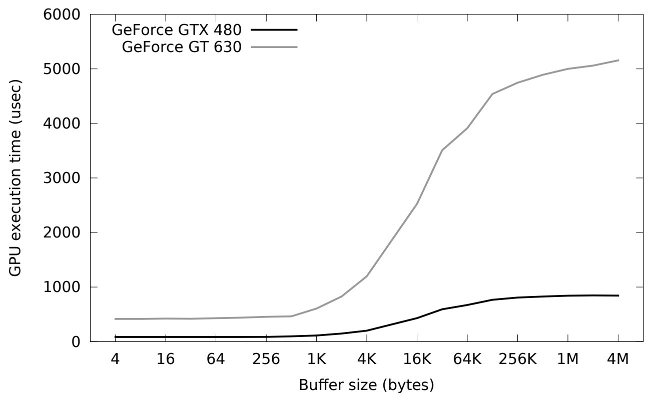


Figure 4: Execution times for low-end (GT630) and high-end (GTX480) graphics cards, when extracting credit card numbers (using the regular expressions of Table 1) for different captured data sizes.

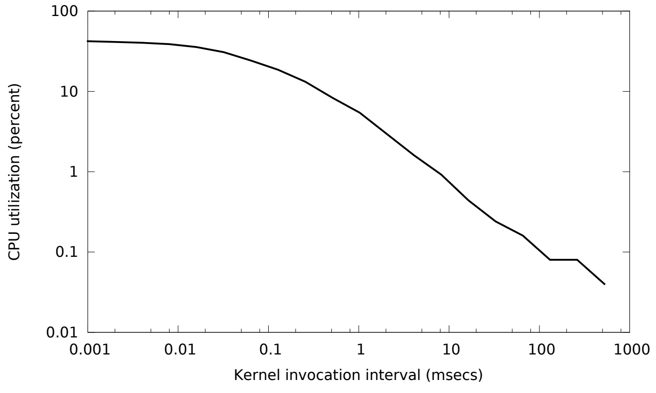
5. COUNTERMEASURES

Current malware analysis and detection systems are tailored to CPU architectures only, and therefore are ineffective against GPU-based malware. Fortunately, however, malicious code that runs on a GPU can be identified in several ways. To properly identify GPU-based malware though, existing defenses need to be enhanced with new functionality for the analysis of GPU machine code.

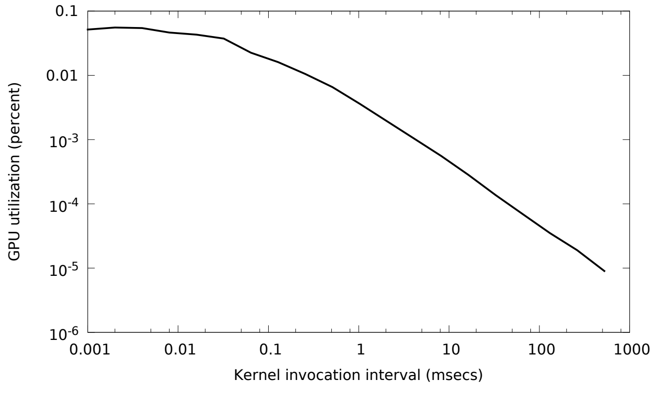
5.1 GPU Code Analysis

NVIDIA recently released `cuda-gdb` and `cuda-memcheck`, two debugger tools for CUDA applications [23]. The goal of `cuda-memcheck` is to provide a lightweight mechanism for checking runtime memory errors. The `cuda-gdb` is capable of debugging in real time a CUDA application running on the actual GPU, similarly to `gdb(1)`. Since version 5.0, `cuda-gdb` can attach to a running process, and inspect the state of the GPU at any point. We note, however, that an attacker could easily strip debug symbols from the malicious code, and significantly complicate its analysis. Still, support for basic debugging of GPU code is a crucial first step toward analyzing GPU-assisted malware binaries. Analogous situations have repeatedly occurred in the past, e.g., whenever popular processor architectures would be extended with additional instructions for floating point or other specialized computations, which malware afterwords exploited for hindering detection and analysis.

An important consideration for malware analysis systems build on top of virtual machine environments [22, 31, 39,



(a) CPU Utilization



(b) GPU Utilization

Figure 3: CPU and GPU utilization of the keylogger for different GPU kernel invocation intervals.

11, 19, 17] is the proper support of GPGPU APIs, in place of basic graphics device emulation. Virtual machine monitors usually provide a virtualization layer between the actual graphics card of the host system, and the emulated graphics card presented to the guest OSes, 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. Recent works have proposed a virtualized environment to provide GPU visibility from within virtual machines [15, 33, 13, 14]. Unfortunately, the purpose of these works is to allow GPU sharing among different applications, using multiplexing and queueing mechanisms, rather than simulating the graphics processors. The latter approach is crucial for tracing the behaviour of a malicious GPU kernel.

5.2 Runtime 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. As a GPU-based keylogger does not need to perform any bulk transfers, it is not clear if this technique could be applied as an effective defense. Alternatively, a possible defense could be based on profiling the GPU utilization or monitoring its access patterns.

6. DISCUSSION

A major limitation of our prototype GPU-based keylogger is that it requires a CPU process to control its execution. The only purpose of the CPU code is to periodically trigger the malicious GPU kernel, an operation that can be implemented with a few machine instructions, resulting in minimal memory footprint. For instance, the CPU binary size of our current prototype is less than 4 KB. This allows an attacker to easily hide the CPU component of the keylogger by injecting its code into the address space of an existing benign user-level process [10, 27, 38].

Another limitation of our prototype implementation is that it requires administrative privileges for initializing the environment required to allow the GPU to monitor the keyboard buffer. However, the code that needs to run with administrative privileges is solely used for acquiring the address of the keyboard buffer and enabling the GPU to access its physical page, and is completely removed afterwards. In contrast to existing rootkits and kernel-level keyloggers, it does not need to hook any code or manipulate any data structures for hiding its presence. The kernel code and data structures remain intact, while the GPU continues to monitor all keyboard activity. As described in Section 3, our prototype uses a loadable kernel module to execute code within the kernel. We should note that this choice was made only for convenience, and the same stealthy approaches that are typically used for the installation of kernel-level rootkits can be employed, e.g., by exploiting a vulnerability and injecting malicious code directly into the kernel.

7. CONCLUSION AND FUTURE WORK

In this paper we presented a stealthy keylogger that runs directly on a graphics processor, allowing it to evade current protection mechanisms that run on the host CPU and are tailored to CPU code. We have implemented and evaluated our GPU-based keylogger on both low-end and high-end NVIDIA graphics cards. Besides recording keystrokes, the architecture of modern graphics processors enables our prototype to benefit from their excess computational capacity for analyzing the captured data. To demonstrate this ability, our prototype uses the streaming processors of the GPU to extract credit card numbers from the captured keystrokes with negligible runtime overhead.

Currently, our GPU keylogger has a small memory footprint on the host memory, and minimal CPU and GPU utilization, about 0.1%. These characteristics can significantly increase its stealthiness and raise the bar against existing defenses.

We conclude that our GPU-based keylogger could be part of a rootkit that, at runtime, would provide a stealthy mechanism for extracting sensitive data from an infected host. Our work clearly demonstrates that additional protection mechanisms are needed to efficiently defend against malicious code executed on graphics processors. As part of our future work, we plan to port our prototype implementation for Windows, and explore similar techniques for performing other malicious activities, including the acquisition of sensitive data, such as cryptographic keys, credentials for web banking accounts, web-camera snapshots, screenshots, and open documents located in the file cache.

8. REFERENCES

[1] Criminals increase their use of keyloggers, according to experts. http://antikeyloggers.com/criminals-increase-use-keyloggers.

[2] Intel SDK for OpenCL Applications 2012. http://software.intel.com/en-us/vcsource/tools/opencl-sdk.

[3] KeyGrabber - Hardware Keylogger - WiFi USB hardware keyloggers. http://www.keelog.com.

[4] KeyGrabber Hardware Keylogger hardware solutions - KeyGrabber Wi-Fi, KeyGrabber USB hardware keyloggers. http://www.keydemon.com.

[5] Keylogger reviews, monitor software comparison, test of best keyloggers 2013. http://www.keylogger.org.

[6] Microsoft's DirectX developer site. http://msdn.microsoft.com/directx.

[7] OpenGL - The Industry Standard for High Perforamnce Graphics. http://www.opengl.org.

[8] Radeons take back graphics card market share. http://techreport.com/news/23482/radeons-take-back-graphics-card-market-share.

[9] AMD. ATI Stream Software Development Kit (SDK) v2.1. http://developer.amd.com/gpu/ATIStreamSDK/Pages/default.aspx.

[10] Anthony Lineberry. Malicious code injection via /dev/mem, March 2009. Black Hat.

[11] U. Bayer and F. Nentwich. Anubis: Analyzing Unknown Binaries, 2009. http://anubis.iseclab.org/.

[12] J. Corbet, A. Rubini, and G. Kroah-Hartman. *Linux Device Drivers, 3rd Edition*, chapter 15, pages 413–414. O'Reilly, February 2005.

[13] J. Duato, A. PeÃśa, F. Silla, R. Mayo, and E. Quintana-Orti. rcuda: Reducing the number of gpu-based accelerators in high performance clusters. In *High Performance Computing and Simulation (HPCS), 2010 International Conference on*, 2010.

[14] G. Giunta, R. Montella, G. Agrillo, and G. Coviello. A gpgpu transparent virtualization component for high performance computing clouds. In *Proceedings of the 16th international Euro-Par conference on Parallel processing: Part I*, EuroPar'10, 2010.

[15] V. Gupta, A. Gavrilovska, K. Schwan, H. Kharche, N. Tolia, V. Talwar, and P. Ranganathan. Gvim: Gpu-accelerated virtual machines. In *Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing*, HPCVirt '09, 2009.

[16] T. Holz, M. Engelberth, and F. Freiling. Learning more about the underground economy: a case-study of keyloggers and dropzones. In *Proceedings of the 14th European conference on Research in computer security*, ESORICS'09, 2009.

[17] 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.

[18] D. Kieras. Using the Keystroke-Level Model to Estimate Execution Times. *University of Michigan*, 2001.

[19] 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.

[20] C. Kruegel, W. Robertson, and G. Vigna. Detecting kernel-level rootkits through binary analysis. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2006.

[21] D. Le, C. Yue, T. Smart, and H. Wang. Detecting kernel level keyloggers through dynamic taint analysis. http://www.wm.edu/as/computerscience/documents/cstechreports/WM-CS-2008-05.pdf, 2008.

[22] 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.

[23] NVIDIA. Compute Unified Device Architecture (CUDA) Toolkit, version 3.2. http://developer.nvidia.com/object/cuda_3_2_downloads.html.

[24] NVIDIA. *Compute Command Line Profiler User Guide, Version 3*, 2011.

[25] S. Ortolani and B. Crispo. Noisykey: tolerating keyloggers via keystrokes hiding. In *Proceedings of the 7th USENIX conference on Hot Topics in Security*, HotSec'12, 2012.

[26] S. Ortolani, C. Giuffrida, and B. Crispo. Bait your hook: a novel detection technique for keyloggers. In *Proceedings of the 13th international conference on Recent advances in intrusion detection*, RAID'10, 2010.

[27] B. Prochazka, T. Vojnar, and M. Drahanský. Hijacking the Linux Kernel. In *MEMICS*, pages 85–92, 2010.

[28] D. Reynaud. GPU Powered Malware. Ruxcon 2008.

[29] A. Seshadri, M. Luk, N. Qu, and A. Perrig. Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, 2007.

[30] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, SOSP '05, 2005.

[31] 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.

[32] S. Shetty. Introduction to spyware keyloggers. www.securityfocus.com/infocus/1829, 2005.

[33] L. Shi, H. Chen, and J. Sun. vcuda: Gpu accelerated high performance computing in virtual machines. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, 2009.

[34] P. Stewin and I. Bystrov. Understanding DMA Malware. In *Proceedings of the 9th Conference on Detection of Intrusions and Malware & Vulnerability Assessment.* DIMVA, Heraklion, Crete, Greece, 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] G. Vasiliadis, M. Polychronakis, and S. Ioannidis. GPU-Assisted Malware. In *Proceedings of the 5th International Conference on Malicious and Unwanted Software (MALWARE)*, 2010.

[37] G. Vasiliadis, M. Polychronakis, and S. Ioannidis. Parallelization and characterization of pattern matching using GPUs. In *Proceedings of the 2011 IEEE International Symposium on Workload Characterization*, IISWC, 2011.

[38] M. Vlad. Rootkits and Malicious Code Injection. *Journal of Mobile, Embedded and Distributed Systems*, 3(2), 2011.

[39] C. Willems, T. Holz, and F. Freiling. Toward automated dynamic malware analysis using CWSandbox. *IEEE Security and Privacy*, 5(2):32–39, 2007.