Offloading Critical Security Operations to the GPU

By

Frank Yi-Fei Wang

under the guidance of

Professor Dan Boneh

Honors Thesis

Submitted in Partial Fulfillment of the Requirements
for the Undergraduate Honors Program
in Computer Science
in the School of Engineering at
Stanford University, 2011

Table of Contents

Abstract

General computing with Graphics Processing Units (GPUs) has taken off the last few years because of their unique ability to perform parallel calculations very efficiently. More developers have started to take advantage of this processor to improve software performance. Similarly, security developers have considered using the GPU to offload calculations as a way to improve the speed and robustness of security software. In this thesis, I outline reasons why this process might not necessarily be secure and provide experiments I have performed to find security flaws in GPU offloading.

Chapter 1
Introduction

1.1 Overview of Parallel Computing

Computer manufacturers are constantly trying to find techniques to improve computational performance. This problem has become more difficult and urgent given the advent of more complex computer programs and systems. For the past few decades, increasing the clock speed of CPUs has been a common and reliable way to solve this problem. However, because of physical limitations on the size of transistors in the CPU and limitations on the circuit fabrication, it is becoming more difficult to improve the clock speed of the CPU, so manufacturers have been looking for alternative methods to increase computational speed.

One method used in the past for supercomputers is to increase the number of processors used in a computer, so the performance is not limited to just the speed of one processor. Instead, multiple processors can be performing calculations at the same time. Using this concept, manufacturers have started to offer processors with multiple computing cores. In fact, it is now difficult to find a personal computer that does not have a multiple core processor.

In order to take advantage of the multiple computing cores, a program has to accommodate parallel execution of computational instructions rather than the traditional sequential programs so that calculations can be done simultaneously or concurrently on the different cores. Many manufacturers are offering multiple core processors as a standard

on personal computers. Understanding parallel computing will, therefore, be important or even required for a computer scientist [SK11].

1.2 Background Information on the GPU

The popularity of graphics-based operating systems like Windows has spurred the development of a new processor to assist the CPU with the additional computational load resulting from the graphical interface. Moreover, companies worked on graphics accelerators to accommodate the rise in applications like Doom and Quake, which use 3D graphics. However, these graphics accelerators primarily relied on hardware optimizations. The concept of the graphics processing unit (GPU) only really emerged when NVIDIA released the GeForce 256. This graphics processor allowed certain graphics computations to be performed directly on the GPU, taking a significant computational load away from the CPU. With such computational power, researchers became increasingly interested in using the GPU to perform non-graphics related (general) calculations.

Initially, standard graphics APIs, DirectX and OpenGL, were very limiting and specific to graphics, and porting code for non-graphics calculations to graphics APIs presented its own challenge. It required representing the calculations as different graphics objects and textures. However, companies like NVIDIA have recently developed better programming environments for general computing, such as Compute Unified Device Architecture (CUDA) [SK11]. Although CUDA can only be used on an NVIDIA GPU, Apple has started to develop a programming framework called OpenCL [Ope] that can be applied across all GPU devices regardless of vendor. Moreover, current GPUs can perform billions of parallel calculations in a second because compared to a CPU, they can execute many more threads in parallel. With these developments, programmers have been using GPUs more frequently for parallel computing and performing CPU-

intensive calculations, and researchers have been searching for different applications for the GPU.

1.3 Overview of General Computing in CUDA and OpenCL

The development of CUDA and OpenCL have made general computing on GPUs much more convenient. Both APIs are built as C extensions and provide functions that allow interaction between the CPU (host) and the GPU (device) and that allow the CPU to control the GPU.

A program written in CUDA or OpenCL is a serial C program that contains parallel code called the kernel. The kernel is run on the GPU when the C program is run. Initially, the CPU runs the code, and then the parent process invokes the GPU by copying the kernel program and shared data to a portion of memory shared between the CPU and GPU. Then, the GPU runs the kernel using the necessary shared data, and the results of the computation are placed into the shared memory, which can later be accessed by the CPU. It is important to note that the kernel cannot run by itself and needs to be invoked by a CPU process, and that the CUDA and OpenCL APIs contain appropriate functions that allow the copying of memory and information between the CPU and GPU.

The remainder of the thesis is organized as follows: Chapter 2 explores the use of GPU for security critical operations, Chapter 3 discusses the risks of offloading anti-virus software operations to the GPU, and Chapter 4 discusses different experiments aimed at discovering weaknesses in the CUDA programming language.

Chapter 2

The Use of the GPU for Security Operations

With the developments that have made performing general computations on the GPU easier, researchers have focused on finding applications for this new source of computational power, especially for security operations, which tend to be computationally intensive. This chapter describes past research done in this area and specifically the desire to offload calculations done by anti-virus software.

2.1 Previous Research

Researchers have been trying to find applications of the GPU to security critical operations because these computations, such as cryptographic operations, tend to be very computationally costly. Moreover, cryptographic operations tend to have a high number of arithmetic operations and sometimes a large demand for memory, which make them ideal operations to be performed on the GPU. Cook et al. made the first attempt to create algorithms for symmetric key encryption by implementing the Advanced Encryption Standard (AES) in OpenGL on GPUs by porting their code into the graphics programming framework in OpenGL [CIKL05]. However, because of the limitations of the OpenGL programming model, the ability of the hardware to perform efficient parallel computations was also limited, and they were hence unable to improve the performance of AES by a significant amount compared to an implementation on the

CPU.

More recently, Yang and Goodman were able to develop algorithms that accelerated AES twofold on the GPU compared to a CPU version [YG07]. Luken et al. found an even faster algorithm, which made the GPU implementation of AES three times faster than a CPU implementation [LB09].

In addition to improving performance of encryption schemes, GPUs have been proposed as a solution to improve performance and robustness of signature-matching Intrusion Detection Systems (IDS), which detect potentially malicious code patterns. Under high packet loads, IDS tends to fail by dropping packets as a way to deal with flow control or by simply crashing. Jacob and Brodley propose a solution to this problem by offloading some packet processing to the GPU [JB06]. They used an open source signature matching IDS called Snort. They ported parts of the code to the GPU, using a language called Cg, which runs with OpenGL. They offloaded the string-processing calculations to the GPU because under high loads, the computations consume about 60 percent of the overall run time. With these changes, they made Snort more robust and faster under high loads compared to a pure CPU implementation.

Unfortunately, although it improves performance for security operations, the GPU can also complicate security, which is the focus of our research problem. Vasiliadis et al. showed that malware writers can use the GPU to create more evasive malware [VPI10]. They created a proof-of-concept implementation that used the GPU to perform unpacking and run-time polymorphism to evade malware detection. They were able to design malware that uses minimal x86 code that merely copies the packed malicious code and bootstraps the process to the GPU. It is important to note that the GPU can run under user privileges, so the malware would work regardless of privilege level. The GPU has thus created the potential for more security attacks and malicious software. The next section discusses using the GPU for anti-virus calculations.

2.2 Offloading Anti-Virus Calculations

Anti-virus software is used to prevent, detect, and eliminate malicious software, using a variety of techniques. As mentioned above, one common technique is to detect malicious code patterns using signature matching. Moreover, in cases where a potential threat or virus has not yet been discovered, such as zero-day threats, the anti-virus software uses heuristics to identify threats. As seen with the Intrusion Detection Systems mentioned above, such software can significantly affect a computer's performance. An anti-virus software has to scan numerous files to detect a potential threat, and the number of files is growing as hard disks are able to accommodate more data. As seen in previous research, the GPU tends to improve the performance of CPU-intensive operations, so there is a desire to offload some of the computations in the anti-virus program to the GPU [JB06]. In fact, the anti-virus makers Kaspersky released a GPU version that they claim runs 360 times faster than a CPU version [Kas].

The next chapter discusses potential risks of this approach.

Chapter 3

Risks of Offloading Anti-Virus Calculations to the GPU

To our knowledge, there has been no research that discusses the security of offloading sensitive operations to the GPU. In particular, we are interested in the process of offloading anti-virus calculations to the GPU. This chapter discusses our concerns and potential risks with offloading security critical information to the GPU.

3.1 Concerns

Although there has been a push to offload security operations to the GPU to improve performance, there has surprisingly been no security analysis ensuring this process is secure. In fact, there is very little documentation or research regarding the security of CUDA or OpenCL, whereas security features and designs of the CPU software are generally well-documented and well-researched. Similarly, there has been no research that shows that operating the GPU under the same security model as the CPU necessarily means that GPU operations are secure. Therefore, the main concern is that moving operations from a processor where security features are generally well-known to a processor whose security design is less known will open up software to numerous attacks.

Specifically, we are focusing on anti-virus software for two main reasons. First, an anti-virus software runs with administrative privileges (the highest kernel privileges), which

makes the software a prominent target for exploits. Second, the purpose of anti-virus software is to detect malware, but if the data returned is unreliable, then basic malware can go virtually undetected.

3.2 Potential Risks

The GPU is subject to several potential risks and may be exploited in various ways. One of the most common risks is denial of service of the GPU. This is of particular interest because the specification for a web version of OpenGL called WebGL specifically warns against this attack [Web]. Moreover, these specifications also provided a proof-of-concept implementation that crashed the operating system because the implemented sent large calculations to the GPU that the hardware spends most of its time rendering. This can also potentially happen with CUDA. Earlier-generation GPUs only allowed one kernel to run at once, and there was no method to terminate the kernel once it began. New-generation GPUs allow more than one kernel to run, but it is possible to flood the GPU with so many kernels that the anti-virus software has no access to the GPU.

Another risk is that since the GPU may contain secure or sensitive data, malicious software may be able to intercept that information. Context Information Security demonstrated that cross-domain images can be intercepted by performing a timing attack to extract pixels when a web browser is using WebGL [For]. It might also be possible for malicious software running on the same machine to extract the image of the computer screen using CUDA or OpenCL. However, our examination of the CUDA and OpenCL APIs showed no explicit functions that can capture the screen image. There might be a timing attack similar to that of the WebGL, but that problem is still open.

Finally, a malicious GPU process can interfere with an anti-virus GPU process, returning

inaccurate data to the anti-virus program. This can cause the anti-virus software to classify malicious software as secure. Our experiments have focused predominantly on this risk, and more details regarding these experiments are described in the next chapter.

Chapter 4

Experiments

In this chapter, we will discuss some experiments we performed to find vulnerabilities in the CUDA programming framework.

4.1 Accessing another process's memory

First, we constructed a very basic experiment. We created two programs that allocated GPU memory and performed simple addition on the GPU. After the first program ran, the other program, which represented the malicious process, would try to access the allocated GPU memory to change the value either before the code sent it to the GPU or change the final result from the GPU. However, this did not work because like the CPU, CUDA virtualizes the GPU memory, and the virtual memory for each program corresponds to different physical memory for the different processes. There was no direct way to access the physical memory of the other process. From this, we also learned that CUDA creates a different context for each running process.

4.2 Forging the CUDA context

Our next experiment focused on trying to forge the context. The goal of this experiment was to trick the CUDA driver into using another process's context for a given process. We first performed a fork, which creates a child process, on the same process and did

the same experiment as above to check if the child process could access the parent process's virtual memory table. That did not succeed, probably because a new context was created for the child process. We suspected that the context was related to some information specific to the process.

Next, we used dtrace, which is the Mac OS X equivalent of a systrace, to trace the system calls that CUDA uses in a program. We found that CUDA creates the context implicitly when a program tries to allocate GPU memory. We also found that CUDA gathers information about a process. Since the experiment with fork did not succeed, we assumed the failure might be related to the ID of the process and threads. Using the Microsoft Detours API to hook the system calls that determine process and thread IDs [HB99], we returned a constant value for the process ID and thread IDs to the driver, and we re-tested our initial experiment. However, this too was unsuccessful, so we hypothesized that the context might contain the path and a fingerprint of the code. We, therefore, ran two instances of the same program with the same code except using randomly generated numbers to see if the driver would confuse the two processes, but it did not.

In trying to find ways for one process to interfere with another, we learned that CUDA uses contexts, but whether another process can forge the context is still unknown.

Chapter 5

Conclusion

With the development of better programming frameworks and GPU, general computing with the GPU has become more widely used in the last decade. Because of the GPU's ability to perform calculations much more quickly and more efficiently in parallel than the CPU can, the GPU has become a very attractive way to reduce CPU usage. As a result, companies and researchers have started to consider using the GPU to perform security critical operations, such as IDS and anti-virus scans, as a way to improve performance and robustness. However, little research has been done to check whether offloading sensitive information to the GPU is secure. In this thesis, I suggest some potential risks that might occur if the GPU were used by security software. I also present some experiments that looked for vulnerabilities in the GPU programming language, CUDA.

The question of whether there is a way to interfere with GPU calculations is still open. Similarly, we are unsure what aspects of CUDA have vulnerabilities and whether the driver runs fully in user space or critical parts run in the kernel. There are numerous avenues to launch attacks on the GPU that have gone unexplored. Before companies offload sensitive data or calculations to the GPU, they, therefore, need to do a better security analysis of the drivers and the programming languages.

Acknowledgements

First, I would like to thank my research advisor Professor Dan Boneh for providing endless support for this project and constantly meeting with me to throw around ideas. I would also like Elie Bursztein for contributing to our discussions and working with me to figure out the correct tools and exploits to try on CUDA.

Bibliography

[CIKL05] D. Cook, J. Ioannidis, A. Keromytis, and J. Luck. Cryptographics: Secret key cryptography using graphics cards. In *Topics in Cryptology â CT-RSA 2005*, volume 3376, pages 334–350. Springer Berlin / Heidelberg, 2005. 4

[For] J Forshaw. Webgl - a new dimension for browser exploitation. http://www.contextis.co.uk/resources/blog/webgl/. 8

[HB99] G. Hunt and D. Brubacher. Detours: Binary interception of win32 function. In *Third USENIX Windows NT symposium*, 1999. 11

[JB06] N. Jacob and C. Brodley. Offloading ids computation to the gpu. In *Computer Security Applications Conference, 2006. ACSAC '06. 22nd Annual*, pages 371 –380, 2006. 5, 6

[Kas] Kaspersky. Kaspersky lab utilizes nvidia technologies to enhance protection. http://www.kaspersky.com/news?id=207575979. 6

[LB09] Desoky AH. Luken BP., Ouyan M. Aes and des encryption with gpu. In *ISCA 22nd International Conference on Parallel and Distributed Computing and Communication Systems*, pages 67–70, 2009. 5

[Ope] Opencl. http://www.khronos.org/opencl. 2

[SK11] J. Sanders and Edward Kandrot. *CUDA By Example*. Addison-Wesley, 2011. 2

[VPI10] G. Vasiliadis, M. Polychronakis, and S. Ioannidis. Gpu-assisted malware. In *Malicious and Unwanted Software (MALWARE), 2010 5th International Conference on*, pages 1–6, 2010. 5

[Web] Webgl specification. https://www.khronos.org/registry/webgl/specs/1.0. 8

[YG07] J. Yang and J. Goodman. Symmetric key cryptography on modern graphics hardware. In Kaoru Kurosawa, editor, *Advances in Cryptology â ASIACRYPT 2007*, volume 4833 of *Lecture Notes in Computer Science*, pages 249–264. Springer Berlin / Heidelberg, 2007. 5