2013

Vulnerability analysis of GPU computing

Michael Patterson
Iowa State University

Vulnerability analysis of GPU computing

by

Michael James Patterson

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering

Program of Study Committee:

Joseph Zambreno, Major Professor

Tom Daniels

Zhao Zhang

Iowa State University

Ames, Iowa

2013

DEDICATION

Soli Deo Gloria

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my thanks to my family and friends. I consider myself the luckiest man in the world, and this is due in large part to the wonderful people who surround me. Thank you.

I would also like to thank my major professor, Joseph Zambreno, for his help and encouragement throughout graduate school. Along with many other professors that I have encountered at Iowa State University, he has always been willing to make time for whatever I request. Thank you.

ABSTRACT

In the past decade Graphics Processing Units (GPUs) have advanced from simple fixed-function graphics accelerators to fully-programmable multi-core architectures capable of supporting thousand of concurrent threads. Their use has spread from the specialized field of graphics into more general processing domains ranging from biomedical imaging to stock market prediction. Despite their increased computational power and range of applications, the security implications of GPUs have not been carefully studied. It has been assumed that the use of a GPU as a coprocessor with physically separate memory space, minimal support for multi-user programming, and limited I/O capability inherently guarantees security.

This research challenges this assumption by demonstrating multiple security vulnerabilities in the current GPU computing infrastructure. Specifically, it focuses on the following three areas:

1. Denial-of-Service by overwhelming the capabilities of the GPU so it is unable to provide responsiveness to the host operating system.

2. Information leakage due to the way that modern GPUs fail to randomize pointers and zero out memory.

3. The use of GPUs to assist CPU-resident malware through obfuscation and unpacking or acceleration of computational intensive tasks such as password cracking or encryption.

Through the use of WebGL and CUDA, we successfully developed a proof of concept attack for the first two vulnerabilities listed above. For the third, we considered several different types of attacks and their implications. In all cases we also suggest possible security measures to fix these vulnerabilities. While the impact of each of these particular exploits is currently hardware and OS specific, current trends in GPU architecture indicate that these problems are only going to rise in importance going forward.

CHAPTER 1. INTRODUCTION

This is an exciting time for the field of Graphics Processing Unit (GPU) computing. As seen in Figure 1.1, a decade ago, the theoretical peak performace of a GPU was roughly the same as that of a CPU [9]. General purpose computing on graphics cards existed, but it was difficult and not widespread. In general, GPUs were used to accelerate graphical displays and not much else. Howerver, over the last decade all that has changed. CPUs ran into the power wall, causing many researchers to conclude that serial computing had reached its performance zenith [5] and could no longer keep up with the projection of Moore's law [23]. CPU manufactures were forced to turn to parallelism to get the needed performance gains. Meanwhile, the performance of GPUs increased at an ever more rapid pace due to the parallelism inherent in modern GPU designs. During much of the last decade, GPU performance has doubled more frequently than 18 months, outpacing even the predictions of Moore's law [14]. Research suggests that this will continue into the future and the architecture of a PC will transform into a heterogeneous model with an increased focus on GPU processing [11].

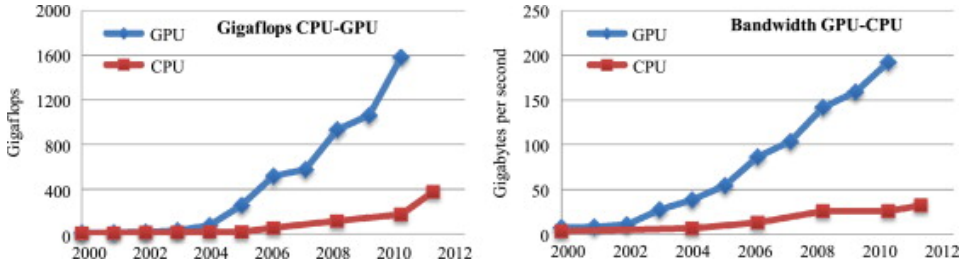

Figure 1.1 GPU and CPU Performance

Along with this massive increase in GPU performance, the range of GPU applications exploded. Three dimensional graphics became pervasive, to the point that GPU acceleration became available within most modern internet browsers [8]. The massive throughput potential

of GPUs was embraced by the high performance computing community, and as of 2011, three out of the top seven supercomputers were GPU-based [19]. In 2007, NVIDIA first released the CUDA toolkit, which allowed GPUs to truly be used for general purpose computing. This sparked a widespread adoption of GPUs as accelerators of any tasks that could be adequately parallelized. Currently, they are being used in the biomedical imageing community [20], the computational geoscience community [37], the financial modeling community [26], and many others.

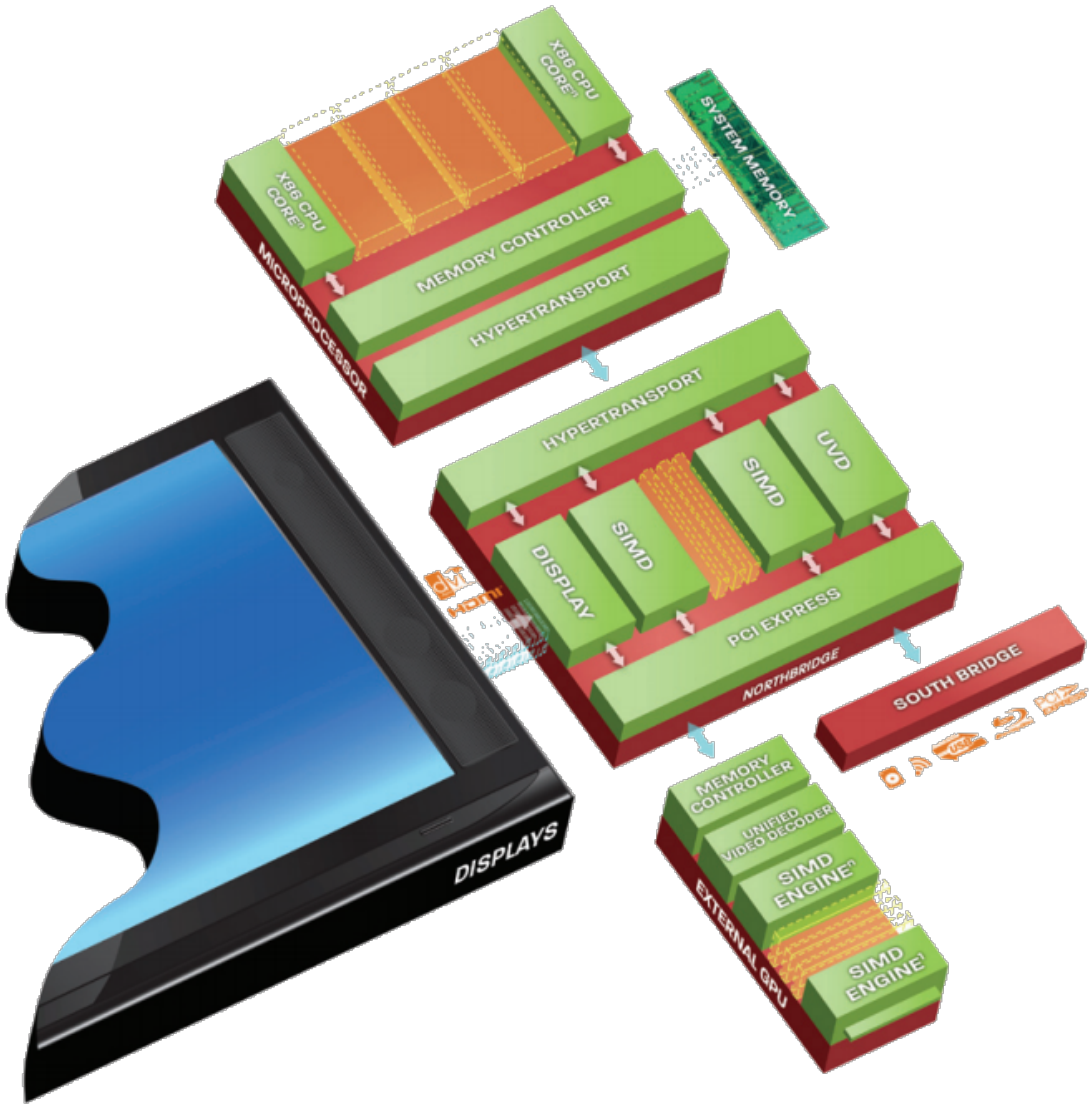

Figure 1.2 Traditional CPU and GPU Architecture

During the last decade, GPU usage was also expanded within the average consumer's PC. Rather than simply being used to accelerate games, GPUs were used for many different tasks. Video encoding and playback were accelerated, desktop windowing systems added advanced effects and used GPUs to render them, and browsers even began to use the GPU to draw the entire webpage.

This integration of tasks between the CPU and GPU naturally resulted in a similar integration of the hardware architecture. This is more thoroughly explained in the AMD Fusion Whitepaper found at [3], but the two main figures from that document show the key concepts. Figure 1.2 shows the traditional architecture, with the CPU and dedicated GPU on different physical chips connected by a bus. Even the integrated GPU is still forced to access the memory controller over a bus. This setup resulted in relatively slow communication between the GPU and key system components such as the CPU and memory controller. However, this system architecture has recently changed. Some systems still have dedicated GPU chips on a separate physical board, but the new direction is to more closely integrate the two types of cores. Figure 1.3 shows the new Accelerated Processing Unit from AMD. This device combines both cores on the same die and gives the GPU first-class access to system memory. Current machines still have a separation of memory between the GPU and the CPU, but a unified memory system is planned for the near future. This discussion has focused on the AMD line of products, but similar changes are being made by Nvidia as well. This trend in increased integration shows the vital role that GPUs currently hold and will continue to hold well into the future.

Along with rapidly growing performance and utilization, GPUs have also skyrocketed in popularity. The term has experienced unprecendented growth in its global search volume over the last decade. Figure 1.4 shows the search history as reported by Google Trends. As can be seen, over the last ten years its popularity has steadily increased and looks to continue this trend into the future. This increased attention is just one of many signs pointing to the fact that GPUs continue to grow in importance.

With all these increases in performance, use cases, and popularity, it would be reasonable to assume that increased attention has also been paid to GPU secuity. However, this is not

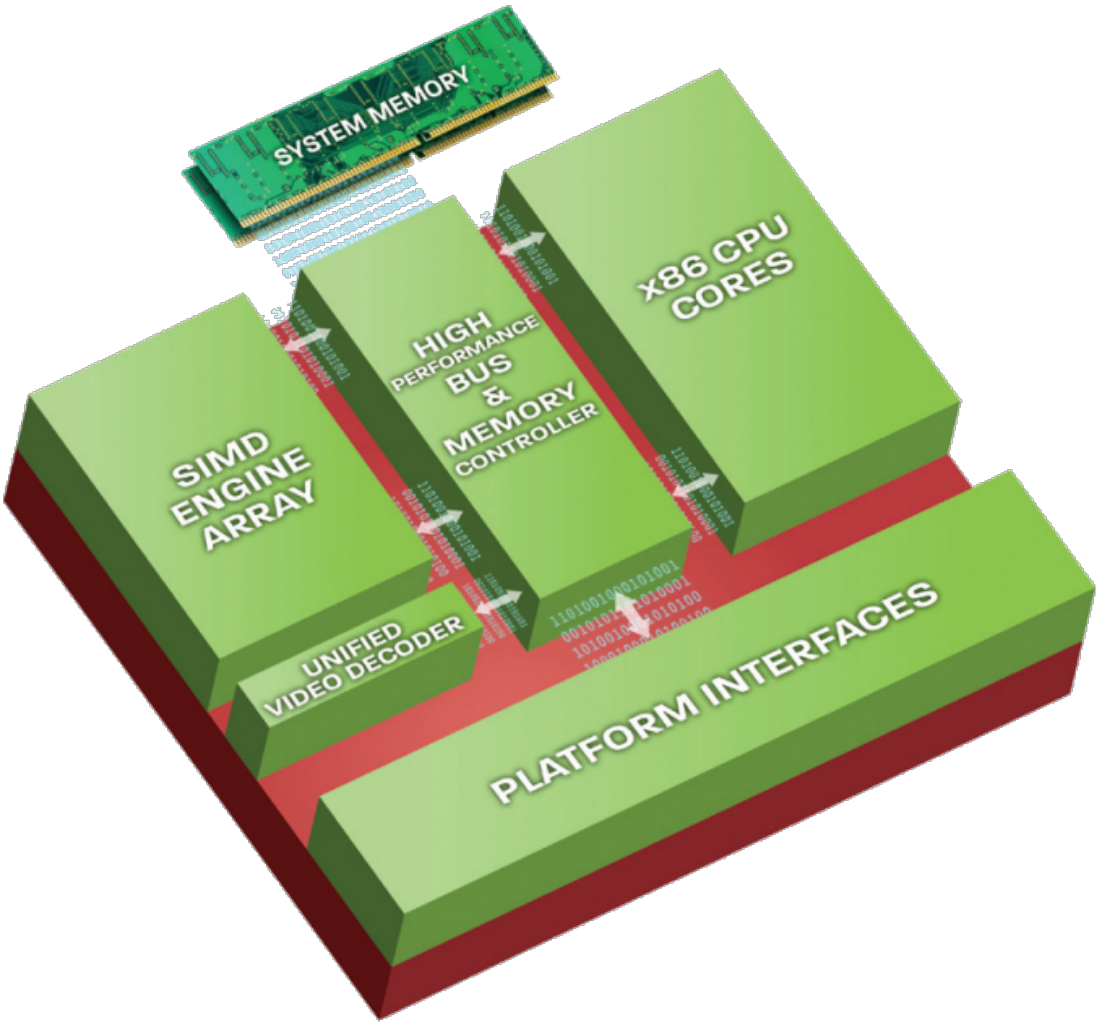

Figure 1.3 New CPU and GPU Architecture

the case. The academic study of GPU vlunerabilities has been almost entirely neglected. One reason there has not been much research in this area is the limitations in the way that GPUs are typically viewed. GPUs in modern PCs are often viewed as coprocessors. The CPU runs the operating system, handles I/O, and dispatches tasks to be executed on the GPU. The GPU processes data, communicates with the CPU, and draws to the display, but beyond that it has no control over the system or any user input.

Even though this type of usage does limit the possible vulnerabilities, this research shows that serious vulnerabilities do exist in the current hardware. We successfully developed a denial of service (DoS) attack that overwhelms the GPU causing the host operating system to

Figure 1.4 Google Trends Search Volume

become unresponsive. Based on the individual system, this results in either a several second freeze followed by a GPU driver reset or a complete system freeze requiring a hard restart. We also discovered several memory vulnerabilities present in all Nvidia GPUs that could be used to leak sensitive information. Additionally, we considered several ways in which GPUs could be used to assist and enhance more traditional malware.

The rest of this paper is organized as follows: Chapter 2 discusses the related scholarly work that has been done in this area and introduces the relevant technologies. Chapter 3 details the denial of service attack, why it is possible, and what steps could be taken to mitigate this vulnerability. Chapter 4 covers the various memory vulnerabilities, again discussing our attack, why it is possible, and how to prevent it. Several other types of GPU-assisted malware are discussed in Chapter 5. Due to the variety, individual attacks are not presented in this chapter, but various concepts are discussed along with possible solutions. Chapter 6 concludes this paper with a summary of our work along with a discussion of possible future work.

CHAPTER 2. RELEVANT TECHNOLOGIES AND RELATED WORK

2.1 GPU Architecture

In order to understand the content of this work, a basic understanding of GPU architecture is necessary. At its most basic level, a GPU is similar to a CPU in that it interacts with memory and executes instructions on data. However, the way in which this is done is very different. Figure 2.1 compares the architecture of a CPU to that of a GPU in a general sense. The important thing to notice is that the GPU dedicates far more resources to ALUs, whereas the CPU dedicates more resources to control logic and caches. This is due to the type of tasks that each device has been specialized to execute. A CPU is optimized to execute sequential tasks. It can execute several heavyweght threads at the same time, with good performance per thread. On the other hand, the GPU is designed to execute thousands of lightweight threads simultaneously, with relatively poor performance per thread. This provides execellent performance when a task can be massively parallelized, but poor performance for serial tasks. The CPU has a more cache-centric memory system to cope with I/O and other tasks with widely varying read times. Inversely, the GPU has little caching but a large and fast ram system to feed data into the many cores.

2.2 Enabling Software Technologies

WebGL is an API from the Khronos Group that brings GPU accelerated graphics to the web browser. By writing some special JavaScript code, a webpage may be made to display a canvas full of content rendered on the client's GPU. The API mimics much of the functionality found in graphics layer APIs such as OpenGL and DirectX. [2] provides a great example of what is possible with WebGL, and the internet is full of resources for both learning WebGL

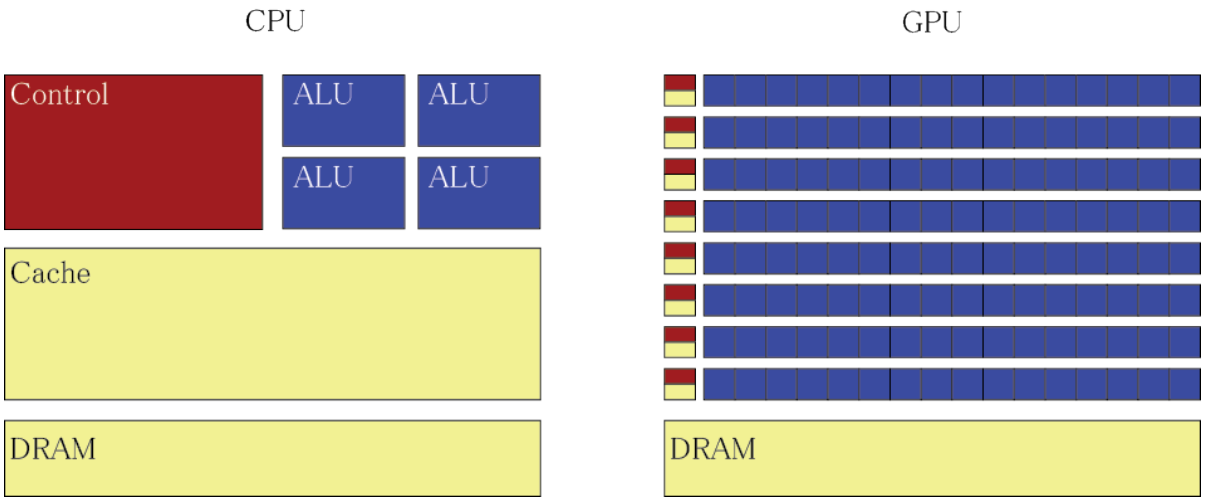

Figure 2.1 CPU Architecture vs. GPU Architecture

and using it. Version 1.0 of WebGL was released in 2011, but it has already seen widespread adoption and support from most major browsers.

In 2007 Nvidia changed the GPU computing landscape by releasing the CUDA toolkit. This set of tools was designed to allow developers to easily perform general purpose computations on a GPU. This was achieved by supporting an extension of C/C++ that specified which portions of a program should execute on the CPU and which portions should execute on the GPU. Nvidia also provided drivers that implemented the CUDA API to support communication between the CPU and the GPU outside of traditional graphics applications. In the years since then, CUDA has gained a huge number of users and developers and become a popular choice for general purpose GPU computing.

2.3 Related Work

As mentioned in the introduction, the amount of scholarly research done in the area of GPU security is surprisingly lacking. As of the time of this writing, a search on Google Scholar for the term "cpu security" returned 186,000 results, but a similar search for the term "gpu security" returned only 15,000 results. Other scholarly databases proved to be equally lacking in resources related to this research area. The vast majority of work done involving GPUs and security investigates the ways that GPU computing can be used to increase security, such as

using a GPU to accelerate intrusion detection systems or anti-virus systems.

A few researchers have discussed the possibility of using a GPU to assist malware. [35] describes multiple techniques used by malware authors to evade detection by virus scanning software. It is then demonstrated how these techniques can be accelerated and improved by using the GPU. The paper focuses specifically on unpacking and run-time polymorphism, but it also includes a very brief discussion of various other techniques that might be possible in the future. [28] also briefly discusses how GPUs could be used to help malware. In this case, the possibility of using a GPU to execute the algorithm for generating a list of command and control servers on a botnet is presented. Additionally, the author points out that at this point disassembly of GPU executable files is impossible, which makes malware analysis of GPU binaries much harder. [30] describes the general GPU landscape. A few paragraphs are dedicated to describing security threats, but most of it is spent summarizing the work done in [35].

None of the vulnerabilities discussed in this thesis are new conceptually, they are just applied to a new domain, GPU computing. Denial of service type attacks have been around for a long time. [15] describes the first denial of service attack that occurred back in 1988. Since then, this type of attack has risen in popularity and ease of execution. Most often it involves sending massive amounts of network traffic over the internet, limiting the responsiveness of a specific website or service. However, DoS attacks aren't limited to this type of scenario. [27] defines a DoS attack as any "explicit attempt to make a network or system unavailable for use," correctly stating that this type of attack can be employed against any type of system. A large amount of research has been published detailing the ways in which DoS attacks are executed over the internet and various mitigation techniques [13] [29] [38], but no formal research exists that applies this type of attack to the modern CPU/GPU system architecture.

Just as DoS attacks have been around for many years in one form or another, so have memory vulnerabilities. [34] attempts to summarize the last 25 years of memory vulnerability exploits, discussing why after a great deal of resarch they continue to be one of the top threats to computer security. These vulnerabilities take many different forms, from the classic buffer overflow vulnerability [7], to format string vulnerabilities [24], to basic information leakaage

between processes. Many different mitigation techniques have been suggested over the years. [32] describes how Address Space Layout Randomization (ASLR) can be used to prevent stack overflow attacks and information leakage. [4] and [10] describe the concept of process isolation, which aims to solve many types of information leakage and memory vulnerabilities.

Many more sources could have been referenced here, as a great deal of research has been done into these types of attacks and mitigations. However, only a few consider the application of these techniques to the current GPU landscape. Even [35] only considers using a GPU to assist malware, it doesn't discuss the vulnerabilities of the GPU architecture itself. The research that follows is the first scholarly work to apply DoS attacks and memory vulnerability attacks to GPUs, and it also expands the discussion of how GPUs can be used to assist malware.

CHAPTER 3. DENIAL OF SERVICE

3.1 Description of the Vulnerability

Most consumer GPUs act as a coprocessor to the host CPU. In this configuration, the GPU executes one task after another in a serialized fashion. Tasks cannot be preempted or halted without reseting the device. In many systems, one of the main tasks that a GPU is responsible for is drawing the OS desktop and interface. This leads to an obvious DoS vulnerability. As shown in Figure 3.1, while the GPU is currently processing a task, it is unable to update the user interface. If the duration of the task is sufficiently long, the user will be faced with an unresponsive system.

3.2 Details of the Attack

In order to take advantage of this vulnerability, an attacker simply has to use the GPU for a task that does not return in a timely manner. There are many ways to do this, and it can easily be done using traditional graphics APIs such as OpenGL and DirectX. However, this section describes multiple attacks using WebGL. WebGL gives web browsers access to GPUs so a website can take advantage of their processing capabilities. This seems like a particularly dangerous attack vector, since these attacks can be launched remotely when an unsuspecting user visits a malicious website.

Three main variations of this attack have been developed. Each one takes advantage of the same vulnerability described above, but in slightly different ways. The following sections describe each variation along with a short code snippet showing the implementation.

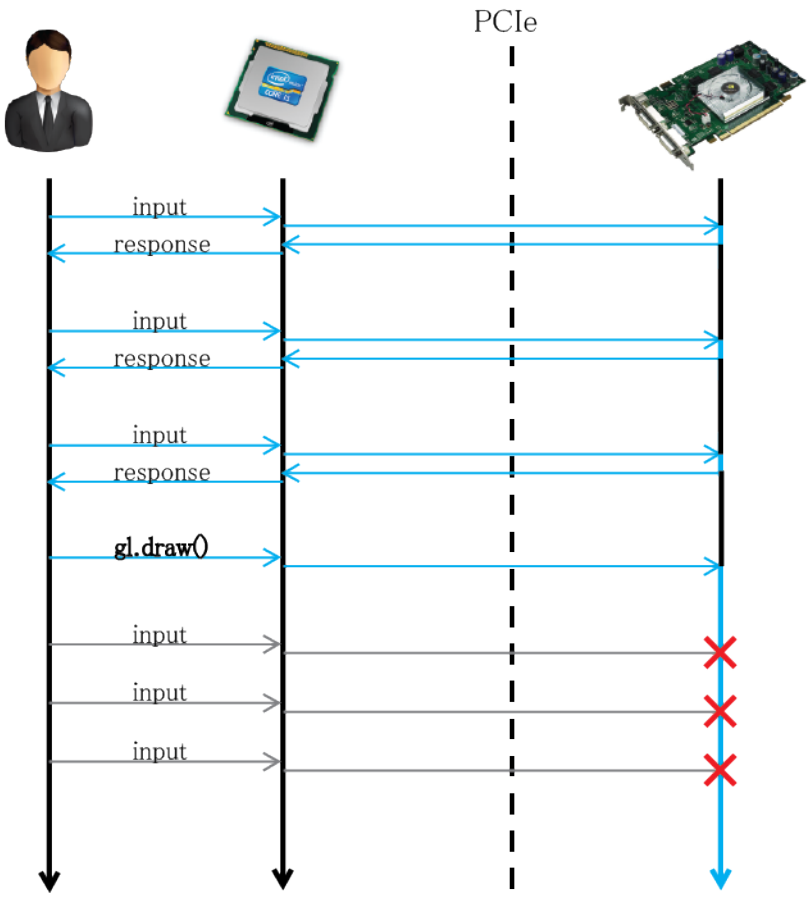

Figure 3.1 GPU Denial of Service

3.2.1 Flooding the `gl.draw` Function

The most straightforward way to perform a DoS attack against a GPU is to simply command it to draw more things than it is capable of drawing in a timely manner. This can be done in any number of ways, but the code in Listing 1 shows a simple attempt at drawing 4,000,000 triangles. Some of the boilerplate WebGL code has been ommitted, but the important details of the exploit should be clear.

```
triangleVertexPositionBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, triangleVertexPositionBuffer);

var numTriangles = 4000000;
var verts = new Float32Array(numTriangles*9);

for(var i=0; i<numTriangles*9; i=i+9) {
        verts[i]   =  0.0; verts[i+1] =  5.0; verts[i+2] =  0.0;
        verts[i+3] = -5.0; verts[i+4] = -5.0; verts[i+5] =  0.0;
        verts[i+6] =  5.0; verts[i+7] = -5.0; verts[i+8] =  0.0;
}
gl.bufferData(gl.ARRAY_BUFFER, verts, gl.STATIC_DRAW);
triangleVertexPositionBuffer.itemSize = 3;
triangleVertexPositionBuffer.numItems = numTriangles*3;
gl.drawArrays(gl.TRIANGLES, 0, triangleVertexPositionBuffer.numItems);
```

Listing 1: Flooding the `gl.draw` Function

This code creates an array of nine values per triangle (three vertices, each needing an x, y, and z coordinate) and then uses the GPU to draw these 4,000,000 triangles. Several implementation details are important to note:

1. It is essential that the drawing is done through a single `gl.drawArrays` call. An alternative approach would be to make a separate call to `gl.drawArrays` for each triangle, but this would miss the point of this attack. Each call is made by the CPU, and it regains control every time the current call returns. By making multiple short calls to `gl.drawArrays`, the OS is able to respond to user input between every call. Only by using the method shown in the above code will the desired unresponsiveness of the OS be achieved.

2. The size of the shape being drawn affects the amount of time that the GPU is unrespon-

sive. If the above code was changed to draw triangles with vertices at (0, 1), (-1, -1), and (1, -1), then the GPU would only be unresponsive for a fraction of the time that it is with the current code.

3. Any type of shape can be used, but the exact number of shapes being drawn is very important. Intuitively, it would seem that the more shapes being drawn the better. However, this is only true up to a point. If the size of the array holding the vertices is too big, the program will crash before the GPU attempts to draw the shapes, and the OS will never become unresponsive. The number of shapes must be picked such that it is small enough to fit in the available memory of the GPU but also large enough that it causes the GPU to become unresponsive when it attempts to render them.

As shown, this code clearly requires modifications which tailor it to the specific hardware being attacked. However, it is fairly trivial to modify this into a more general purpose attack. The WebGL API and others provide functionality to query the characteristics of the host hardware to determine its memory size and processor capabilities. These values could then be used to dynamically choose an appropriate size and number of shapes to draw. In this way, the above code can be used to effectively attack any user that visits a website hosting it.

3.2.2 Flooding the Vertex Shader

Another possible attack vector is the vertex shader. A vertex shader is the stage of the graphics pipeline responsible for processing each individual vertex. It handles the translation from 3D coordinates to 2D screen coordinates and processes any desired modifications to vertex attributes. Shaders are a little different than typical WebGL code in that they are precompiled by the host and then loaded onto the GPU prior to a WebGL program's execution. In order to leverage the vertex shader in a DoS attack, the attacker needs to craft a shader program that cannot be quickly executed on the GPU. There are many ways to do this, but a simple attack can be done by including an infinite loop in the shader code. The shader code in Listing 2 demonstrates such an attack. Note that the WebGL code is not shown, but this shader code can be incorporated into a WebGL application easily. Simply drawing a few shapes using this

shader will result in a DoS attack.

```
<script id="shader-vs" type="x-shader/x-vertex" >
attribute vec3 aVertexPosition;
uniform mat4 uMVMatrix;
uniform mat4 uPMatrix;

void main(void) {
        float val;
        for(float i=0.0; i!=0.5; i+=1.0)
                val = val+0.000001;
        gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPosition, 1.0);
}
</script>
```

Listing 2: Flooding the Vertex Shader

This code implements a vertex shader containing an infinite loop, since the value `i` will never be equal to `0.5`. As described in [18], support for looping constructs in shaders is very limitied, so the code must be written very specifically to cause an infinite loop while still meeting the constraints. Several implementation details are important to note:

1. There is no guarantee that `while` loops in shaders will be supported on a given platform. The use of a `for` loop ensures that this code will correctly execute on the widest range of targets.

2. Loops are not allowed to have variables in the conditional expression other than the loop index. This was meant to prevent infinite loops, but the above code shows how an infinite loop can still be achieved within these constraints.

3. Due to the way that shaders are compiled and used on the GPU, there is a maximum size constraint that differs with each model of GPU. One could attempt to write a shader

that performs many instructions linearly without iterations, but it's likely that this size constraint would be reached before an effective DoS attack was achieved. The implementation presented above results in a short code segment, so this limitation is not a problem.

4. The variable calculated in the loop is actually used in the calculation of the vertex position. While this is not absolutely necessary at the time of this writing, it's likely that in the future compilers will advance to the point of optimizing out code that has no effect.

3.2.3 Flooding the Fragment Shader

Much like the vertex shader attack vector described above, the fragment shader could also be used in a DoS attack. The fragment shader is responsible for computing color and other attributes for every fragment. In order to leverage it in a DoS attack, an infinite loop can be inserted into the shader code. The code in Listing 3 shows a simple way to do this. Note that the WebGL code is not shown, but this shader code can be incorporated into a WebGL application easily. Simply drawing a few shapes using this shader will result in a DoS attack.

```
<script id="shader-fs" type="x-shader/x-fragment" >
precision mediump float;

void main(void) {
        float val = 1.0;
        for(float i=0.0; i!=0.5; i+=1.0)
                val = val+0.000001;
        gl_FragColor = vec4(val, val, val, 1.0);
}
</script>
```

Listing 3: Flooding the Fragment Shader

This code implements a fragment shader with an infinite loop. The structure of this code is

very similar to the code for the vertex shader that was already discussed, but the GPU handles the shaders differently since they are in different positions in the graphics pipeline. Again, support for looping constructs is very limited, so the same details mentioned in the discussion of the vertex shader code apply here as well.

3.3 Impact

WebGL is a fairly new technology, so it is not yet supported on all platforms. Modern versions of Firefox and Chrome support WebGL, but even the newest version of Internet Explorer does not. The newest version of Safari does provide WebGL support, but it must be manually activated, and Safari ships with it deactivated by default. Currently no major mobile browsers offer WebGL support, but this situation will likely change in the very near future. [6] demonstrates that WebGL support is already built into the mobile version of Safari, but developers do not yet have an official way to activate it. The situation is similar for the mobile versions of Chrome and Firefox as well, since the desktop browser codebase includes WebGL support, it just has not yet been activated for the mobile versions. All signs point to the fact that in the near future WebGL will be supported on both Android and iOS devices, which is a large majority of the mobile device market.

Since WebGL is currently only fully supported on desktop versions of Firefox, Chrome, and Safari, these browsers were used for testing. Each of the following three sections summarizes the impact of one of the attacks and includes a table showing how it affected various combinations of OSs and video cards. It turned out that each browser responded to the attacks the same way, so the specific browser being used is not included in the following sections. Keep in mind that the system freezes caused by these attacks are total OS freezes, not just web browser freezes. The entire OS is unresponsive and nothing gets redrawn on the display.

3.3.1 Impact of Flooding the `gl.draw` Function

Table 3.1 contains the results of testing the first attack. In most cases, the system froze for 10-15 seconds, the screen went blank, and then the desktop came back up with a message saying that the graphics driver was reset. Figure 3.2 shows this message on a Windows 7 machine

Table 3.1 Results of Flooding the `gl.draw` Function

	Nvidia GPU	ATI GPU	Intel GPU
Windows XP	Total system freeze.	System freeze, then GPU recovery message	Not tested
Windows 7	System freeze, then graphics driver reset.	System freeze, then graphics driver reset. Occasional total system freeze.	System freeze, then graphics drvier reset.
Mac OS X	Total system freeze.	Total system freeze.	Not tested
Red Hat Linux	System freeze, then graphics driver reset.	System freeze, then graphics driver reset.	System freeze, then graphics driver reset.

with an Nvidia GPU. Some cases, most notably machines running Mac OS X, resulted in a complete system freeze that required a hard reset. None of the tested systems responded to this attack without freezing for at least several seconds.

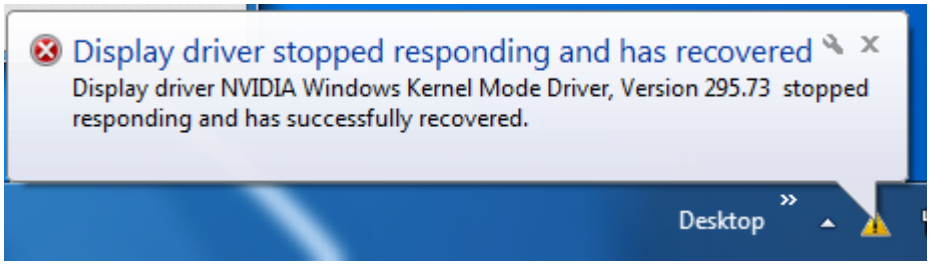

Figure 3.2 Windows 7 GPU Reset Message

3.3.2 Impact of Flooding the Vertex Shader

Table 3.2 contains the results of testing the vertex shader attack. In most cases, the system froze for 10-15 seconds, the screen went blank, and then the desktop came back up with the same message as the first attack. In fact, Windows XP, Windows 7, and MAC OS X all responded in the exact same way as the first test. However, Linux fared much worse in this attack. Flooding the vertex shader resulted in a complete system freeze regardless of GPU vendor. It's not entirely clear why this different behavior is happening, but it suggests that there is either a bug in the recovery code or that the recovery is dependent on the specific stage

Table 3.2 Results of the Vertex Shader Attack

	Nvidia GPU	ATI GPU	Intel GPU
Windows XP	Total system freeze.	System freeze, then GPU recovery message	Not tested
Windows 7	System freeze, then graphics driver reset.	System freeze, then graphics driver reset.	System freeze, then graphics driver reset.
Mac OS X	Total system freeze.	Total system freeze.	Not tested
Red Hat Linux	Total system freeze.	Total system freeze.	Total system freeze.

Table 3.3 Results of the Fragment Shader Attack

	Nvidia GPU	ATI GPU	Intel GPU
Windows XP	Total system freeze.	System freeze, then GPU recovery message	Not tested
Windows 7	System freeze, then graphics driver reset.	System freeze, then graphics driver reset.	System freeze, then graphics driver reset.
Mac OS X	Total system freeze.	Total system freeze.	Not tested
Red Hat Linux	Hang, then graphics driver reset.	Hang, then graphics driver reset.	Hang, then graphics driver reset.

of the graphics pipeline in which the looping occurs.

3.3.3 Impact of Flooding the Fragment Shader

Table 3.3 contains the results of testing the fragment shader attack. In almost every case, it resulted in the same behavior as the first attack. The one exception is a Windows 7 machine with an ATI graphics card. This setup occasionally resulted in a complete system freeze when the first attack was performed. However, the fragment shader attack never resulted in a complete system freeze for this type of system.

Some clear patterns emerge when looking at this data. First, the Windows machines were very consistent throughout all three attacks. For Windows XP, an Nvidia machine always froze

completely and an ATI machine always hung for several seconds and then recovered. Windows 7 machines always hung for a few seconds and then recovered, regardless of video card. Mac OS X froze completely under every attack, and Red Hat Linux froze completely under the vertex shader attack. This variety of response likely results from the complex system that is used to render WebGL content using the GPU. Even though web browsers and driver versions are not included in these tables, these two factors along with operating system type and specific GPU all play a role in the behavior of a PC subjected to one of these attacks. At the time of this research all web browsers and drivers lead to the same behavior, but this may not always be the case. One particular browser could implement one of the suggested mitigations or a graphics driver could change the way that the driver reset is handled. In this case, there would be even more variables in determining the response of a given system.

3.4 Mitigations

3.4.1 Existing Mitigations

As mentioned in the previous section, some mitigations for this vulnerability already exist. Windows Vista, Windows 7, and Linux all have timers to detect when a GPU is unresponsive. When this is detected, the OS will reset the video driver to restore functionality. However, testing the previous exploits revealed that this mitigation does not always function as well as advertised. [22] shows the Windows documentation claiming this timer is two seconds, but testing of the previously mentioned attack typically resulted in a freeze of roughly one minute. Additionally, after a certain number of timeouts (default 5), Windows will crash entirely. The Linux detection typically worked in a more timely manner, but in the case of the vertex shader attack the GPU fault was never detected and the system was entirely unresponsive. At the time of this writing, Mac OS X had no mitigations for this type of attack. Overall, the existing mitigations are highly inadequate.

3.4.2 Suggested Mitigations

The unfortunate thing about many DoS vulnerabilities is that they are often inherent to the way that the system is designed to function. As is so often the case, there isn't really a way to completely eliminate this vulnerability while still providing a useful service to the user without a major redesign of the system. Oftentimes it is deemed enough to simply make the DoS very difficult to execute effectively. In this case, there seem to be two main options for mitigation, depending on the willingness of the vendor to change the design. The first option is to do some type of software filtering on the code prior to executing it on the GPU. Either the web browser's implementation of WebGL or the GPU driver could do some static analysis to attempt to estimate the runtime of the GPU function calls prior to allowing the application to execute. This could be done by analyzing the number of shapes being drawn and the complexity of the shaders. If the estimated runtime was greater than a certain value, then it could prevent the application from launching. This is not an ideal solution, since static code analysis is very difficult and sometimes the runtime cannot properly be estimated. However, a new WebGL engine from Google known as ANGLE [16] already validates shaders, so it seems reasonable to think that this functionality could be extended to prevent some of the simpler DoS attacks.

Another option would be to redesign the architecture of the GPU to support simultaneous execution of different tasks. If this was done correctly, some GPU resources could always be responsible for the necessary OS functions, even if another task on the GPU was not responsive. This mitigation technique is much more robust than the previously described static code analysis, but it also requires major changes to the existing GPU hardware. Additionally, the ramifications of this change are vast and unknown. It's likely that the vendors would rather put up with this DoS vulnerability than implement such a major change.

At the very least, the current mitigation strategy of detecting GPU faults and reseting the adapter should be adopted by all major operating systems. It should also be more thoroughly tested to ensure that a complete system crash never happens. The timing should also be improved so a user is not faced with an OS that is unresponsive for a whole minute prior to the driver reset occurring.

CHAPTER 4. MEMORY VULNERABILITIES

4.1 Description of the Vulnerabilities

4.1.1 Process Isolation and ASLR

In order to understand the memory vulnerabilities present on modern GPUs, it's important to first understand two major security techniques being used by modern PCs. Process isolation is a cornerstone of computer security. The basic idea is that each process on an operating system should be protected from all other processes. Specific channels may be provided to support inter-process communication, but otherwise a process should never be able to interfere with the execution of another process or read/write its memory contents.

Address Space Layout Randomization (ASLR) is a security technique that attempts to make several memory vulnerabilities much harder to exploit. It does this by randomly positioning various data areas within a process' address space. For example, consider the simple C program in Listing 4.

```
#include <stdlib.h>
#include <stdio.h>
void main(void) {
        int *p = (int*)malloc(sizeof(int));
        printf("%d\n", p);
}
```

Listing 4: CPU ASLR Test

This program uses `malloc` to get a pointer to the heap. The value of this pointer is

then printed out as an integer and the program exits. Running this program consecutively demonstrates the ASLR that is implemented on all modern operating systems. As shown in Figure 4.1, the exact same memory allocation will result in different addresses during different program executions due to the randomization of the heap. This improves security in many ways and makes various buffer overflow attacks very difficult to execute. It also protects against information leakage when process isolation is not fully implemented.

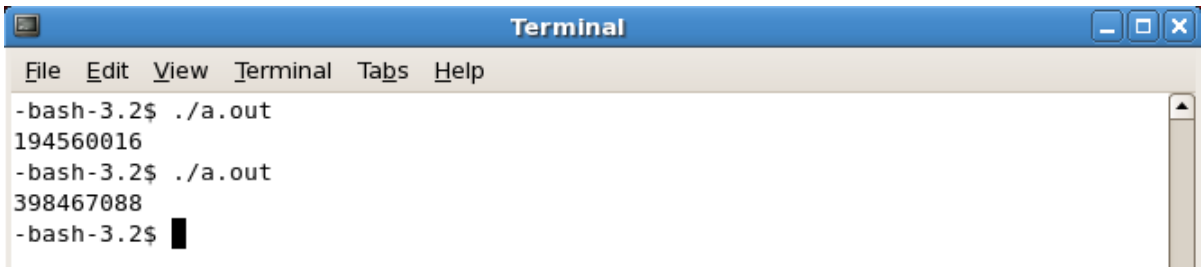

Figure 4.1 CPU ASLR

4.1.2 CPU/OS Implementation

In general, most modern CPUs and OSs implement process isolation through a combination of virtual memory and ASLR. Figure 4.2 shows how virtual memory addresses can be used to guarantee memory isolation for each process. Every virtual address used in a process gets translated through a page table into a unique physical memory address. In this way, process A and process B can both use memory address `'0'` without any collision or memory leakage.

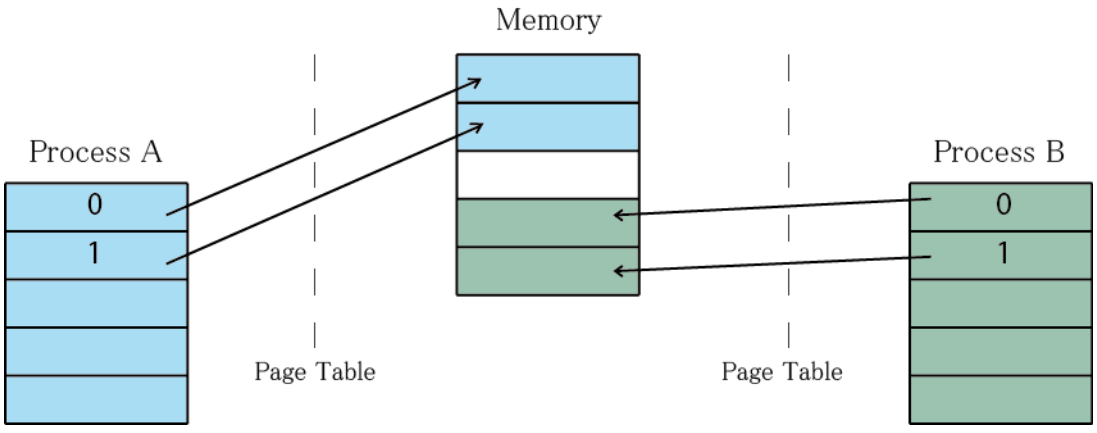

Figure 4.2 CPU Memory Isolation

Additionally, ASLR has been implemented by all of the major operating systems for several years. Even if process isolation did not exist, information leakage between processes would be difficult due to ASLR. The location of the desired data would have to be guessed each time the attack was executed. This is possible to do, and brute force attacks exist, but it is much more difficult than simply knowing where key data areas are within a program's address space.

4.1.3 GPU Implementation

Current GPUs do not implement either virtual memory or ASLR. Figure 4.3 shows what memory access looks like on modern GPUs due to the lack of virtual memory. Memory address `'0'` is the same physical memory location for every process. If process B uses a pointer to an address that has previously been used by process A, it will access the same physical memory that was used by process A. This may not seem to be a problem since current GPUs only execute one process at a time, but when combined with the following vulnerabilities it can be exploited.

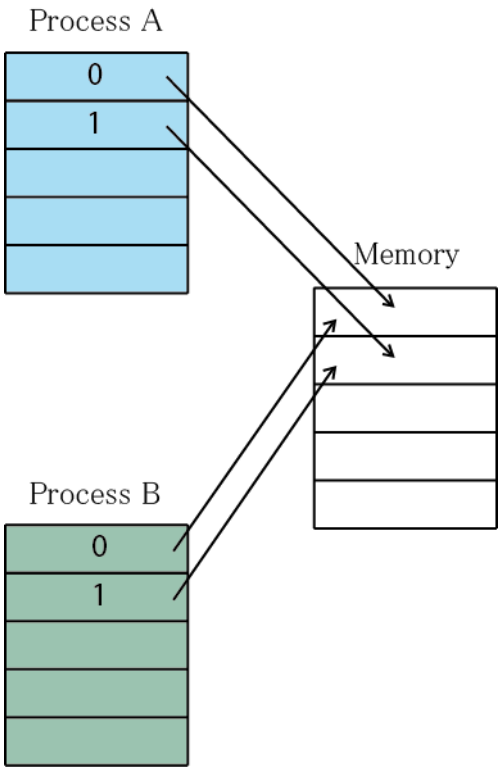

Figure 4.3 GPU Memory Access

ASLR is currently not implemented on any modern GPUs. Pointer allocations consistenly return the same addresses, which indicates that key data areas are not randomized. Consider the code in Listing 5 which is an adaptation of the ASLR demo program in the previous section.

```
#include <stdio.h>
int main() {
        int *a_device;
        cudaMalloc((void **)&a_device, sizeof(int));
        printf("%d\n", a_device);
        return 0;
}
```

Listing 5: GPU ASLR Test

This program uses `cudaMalloc` to get a pointer to the global memory on the GPU. The value of this pointer is then printed out as an integer and the program exits. Running this program consecutively demonstrates the lack of ASLR on a GPU. As shown in Figure 4.4, the same memory allocation will result in the exact same addresses during different program executions.

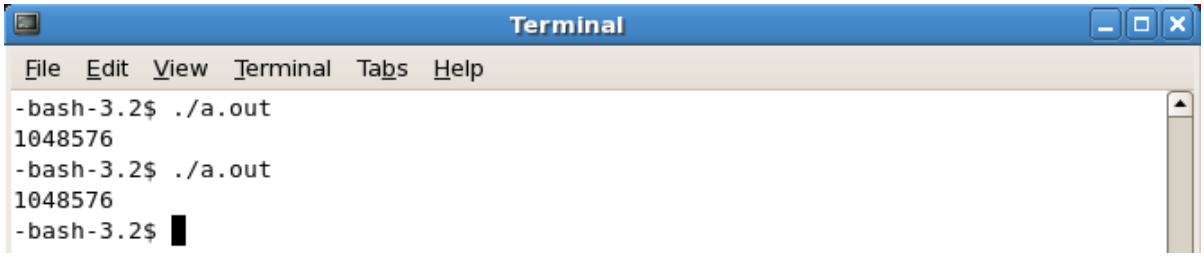

Figure 4.4 CPU ASLR

Additionally, GPUs do not zero out memory prior to allocation or after deallocation. Combined with the lack of virtual memory and lack of ASLR, this results in a clear vulnerability for information leakage. Pointer allocations are not randomized, so the addresses of various data can be determined. Virtual memory does not exist, so these addresses access the same physical memory accross processes. This memory does not get cleared, so the data remains in

memory even after program execution ends.

4.2 Details of the Attack

Based on these vulnerabilities, there are many ways that information leakage attacks could be done. In this research, we present a proof-of-concept attack that demonstrates how these vulnerabilities can be exploited to leak information from one CUDA program to another. For this attack, two CUDA programs have been developed. The first, `savekey.cu`, consists of the code in Listing 6.

```
#include <stdio.h>
int main() {
        float *a_device;
        float a_host = 95.0;
        cudaMalloc((void **)&a_device, sizeof(float));
        cudaMemcpy(a_device, &a_host, sizeof(float),
                                cudaMemcpyHostToDevice);
        printf("The value %f was written to the device.\n", a_host);
        cudaFree(a_device);
        return 0;
}
```

Listing 6: `savekey.cu` CUDA code

This program allocates memory on the GPU, stores the value `95.0` to this memory, prints the value that was stored, and then releases the memory on the device. In this testing scenario, this program is the victim that leaks information to the attacker. The second, `getkey.cu`, consists of the code in Listing 7.

```
#include <stdio.h>
int main() {
        float a_host;
        float *a_device;
        cudaMalloc((void **)&a_device, sizeof(float));
        cudaMemcpy(&a_host, a_device, sizeof(float),
                                cudaMemcpyDeviceToHost);
        printf("The value %f was retrieved from the device.\n",
                                a_host);
        cudaFree(a_device);
        return 0;
}
```

Listing 7: `getkey.cu` CUDA code

This program allocates some memory on the GPU, copies whatever is stored at this location back to the CPU, prints the value that was recovered, and then releases the memory on the device. The important feature of this program is that the location in the GPU memory that is being read from is never initialized. In this testing scenario, this program is the attacker, and the goal is to recover the value that was stored on the GPU by the previous program. If this program recovers the correct value, in this case `95.0`, the attack is successful and the information leakage vulnerability is demonstrated.

4.3 Results

As expected based on the vulnerabilites, the first CUDA application leaked data to the second CUDA application. This was tested by a three step process: first `getkey.cu` code was executed to demonstrate that the original memory did not contain `95.0`. Next, the `savekey.cu` code was executed to simulate the usage of the GPU by an outside program. Finally, the `getkey.cu` code was executed a second time to demonstrate the succesfull recovery of the

value used in the previous program's execution. Figure 4.5 shows the output of these three steps:

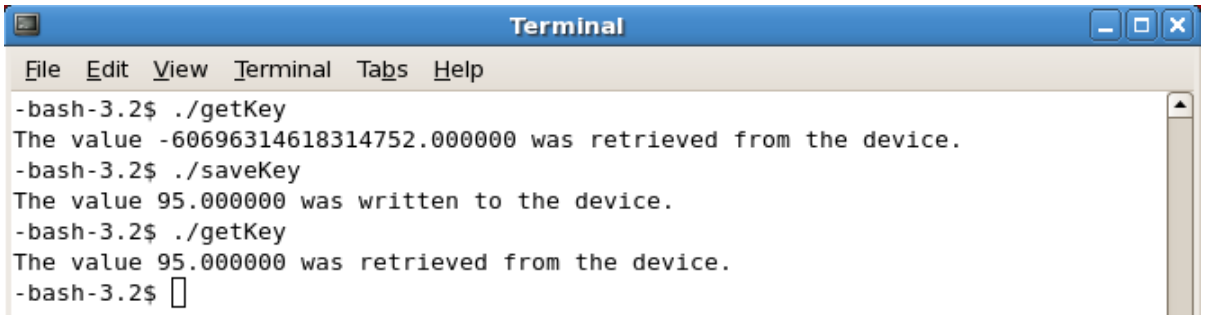

Figure 4.5 Information Leakage between two CUDA Applications

This attack was successful since the information leaked from the `savekey` program's execution to the `getkey` program's execution. This succesfully demonstrated the information leakage vulnerability between CUDA applications. Futhermore, this attack was successful across multiple users and login sessions. There is no automatic mechanism to clear the GPU memory, so it's possible that information can stay in GPU memory for an indefinite amount of time, able to be read by anyone who knows where to look.

4.4 Implications

This proof-of-concept attack is harmless, but it has serious implications. GPU hardware offers no memory protection, and at this point most software doesn't either. What this means is that any information that is processed on the GPU is vulnerable to information leakage. A decade ago this might not have been too serious since games were the only applications that did much processing on the GPU, but that's no longer the case. GPUs are already used for very important processing, and the trend is for more and more to be done on GPUs. [25] describes countless CUDA applications in scientific research, commercial ventures, and government contracts. [17] describes how Google Chrome now has an option to render entire webpages using the GPU, not just WebGL content. From valuable research data to personal bank account information, all this sensitive material is vulnerable if individual software security measures are not used.

This CUDA attack shows how to simply read the contents of global memory, but it could easily be extended to intelligently search through the memory contents looking for specific types of data. [36] demonstrates how statistical methods can be used to recover chunks of information from a corrupted hard drive, and this same method could be applied here. In fact, the GPU could even be used to accelerate this statistical analysis and quickly find valuable types of data. Additional research could be done to find patterns in locations of data storage, and this could be used to more efficiently extract valuable information. An attack like this could be attached to existing malware to collect information from the GPU along with keystroke logging and screen capturing. With this fundamental vulnerability, the possible applications are vast.

4.5 Mitigations

4.5.1 Existing Mitigations

Some GPU programming APIs have already implemented mitigations for these types of attacks. WebGL clears the contents of memory prior to allocating that space, so it is not possible to use this as the attack vector. However, since it does not clear the memory after it has been used, WebGL applications are still vulnerable to information leakage as shown previously in this chapter. The CUDA driver does only allow access to addresses that have already been allocated to a program, but it doesn't prevent you from allocating all of the global memory and reading from it. Clearly the existing mitigations are not enough to protect against these attacks.

4.5.2 Suggested Mitigations

A simple mitigation for this vulnerability would be a GPU driver that automatically cleared the contents of memory whenever it was done being used. This would defeat both of the attacks laid out in this chapter. However, there are legitimate reasons why this is not currently done. This operation adds an overhead to program execution. While these types of overheads may be tolerable in most OSs, the focus on GPU performance makes them less acceptable in this context. It's a possibility that this type of security will be implemented in the future, but it's

understandable why it is not already. However, there is nothing stopping the implementation of this same type of security by the developer. API programmers could write library functions in such a way that when memory is released it is automatically cleared first. Even if this doesn't happen, application developers could manually clear memory contents prior to releasing them. The most robust solution is obviously for this to be implemented at the lowest level possible, but even at the application level it will at least prevent information leakage from that application. Possibly the best solution is for GPU vendors to provide this security as an option that can be enabled when desired.

Another possible mitigation technique would be a GPU architecture that supports virtual memory. Currently this idea is not implemented due to the fact that only one process can be executing at a given time, but it's possible that in the future this might change, and then the idea of virtual memory makes a great deal of sense. With virtual memory implemented, process isolation could be easily enforced and ASLR could be implemented in the same way that it currently is on modern OSs.

CHAPTER 5. GPU-ASSISTED MALWARE

Along with the specific vulnerabilities described in this work, there are several interesting ways in which a GPU can be used to assist traditional malware. Chapter 2 mentioned several possibilities that have been discussed in existing research. In the following sections we present a brief survey of these possibilities and several new considerations. In general, the major benefits of utilizing a GPU in this context are the following: Code executing on a GPU is not scannable by the CPU. Additionally, at this time no security software does any analysis of GPU binaries. These factors combine to mean that current anti-virus software offers no protection against malicious CUDA code. CUDA applications do not need any elevated priviledges, so any user can launch them. Finally, CUDA applications are inherently more stealthy, since most users are much more likely to monitor/notice a spike in CPU usage but not one in GPU usage.

5.1 Possible Attacks

5.1.1 Unpacking and Run-time Polymorphism

[35] is the most thorough research done in the area of GPU-assisted malware. It describes several strategies used by malware authors to prevent detection, focusing on unpacking and run-time polymorphism. It then demonstrates how these strategies can be improved by using CUDA to execute key functions on the GPU. The GPU is able to greatly improve the unpacking technique due to its computational performance and the inability of analysts to inspect CUDA binary files. Complex encryption schemes can be used to hide the contents of the malware since the power of the GPU allows faster decryption than that of a CPU [21]. The host code simply has to load the encrypted data onto the GPU and then call the GPU function to unpack the code. This minimizes the amount of host malware code that is accessible to security researchers,

since the GPU binary code is essentially a black box.

This functionality would offer very little real benefit if some type of run-time polymorphism was not also implemented. Once the unpacking phase is complete, the original malware code is in the host memory and vulnerable to any type of analysis. For this reason, many malware authors implement run-time polymorphism by only unpacking one chunk of code at a time. This technique could also be implemented on the GPU to improve results. The benefit would again be malware that is much harder to reverse engineer and detect.

5.1.2 Direct Memory Access

[31] mentions the possibility of attacking a system by taking advantage of Direct Memory Access (DMA) to access memory that otherwise would be protected. While not specifically mentioned in this paper, the extension of this idea to GPUs represents a large vulnerability. DMA allows the GPU to access system memory independently of the CPU. Since the CPU does all the memory protection, this DMA would bypass any type of memory security and allow unlimited access to system memory. [12] describes how this type of attack has been successfully implemented in the past using a network card. However, our research in this area suggests that it is currently impossible to use a GPU to perform such an attack. This is due to the way that DMA is implemented in CUDA. In order to use DMA, certain asynchronous copy functions are used. The CPU is still responsible for allocating host memory, and this pinned memory is passed to the GPU for use in DMA. This prevents the vulnerability, since the CPU still controls the memory access and the protections are still in place. The GPU can only use DMA to access the memory that was already given to it by the CPU, not any of the system protected memory. It's possible that in the future the GPU will have more control over its DMA, but at the moment this type of attack is not possible.

5.1.3 Framebuffer and Screen Capture

[35] also mentions the framebuffer as a potential attack vector. The framebuffer resides in the GPU memory, so in theory a GPU program could be designed to access the framebuffer for malicious reasons. This program could grab the contents and act as a stealthy screengrabber.

It could also be used in a sophisticated phishing attack by analyzing the current display and modifying certain regions of it. For example, it could analyze the address bar of a web browser and replace the real address with a fake one. This would lead to very hard to detect phishing attacks.

These ideas are interesting concepts, but our research suggests that they are not possible to implement using current GPU programming APIs. While the framebuffer does reside on the GPU, none of the APIs expose direct access to it to the programmer. CUDA cannot directly read from the framebuffer or write to it. It's possible that this functionality may be added in the future, but currently it does not exist.

However, a GPU would still be useful in a related attack. It could still be used to assist screengrabbing malware in multiple ways. First, it could analyze the captured screenshot to determine if it contains any useful information. This type of image analysis is computationally intensive, so it would be perfect to execute on the GPU without taking up CPU cycles and noticeably slowing the machine. The GPU could also be used to compress or encrypt the image prior to its transmission. These are also operations that map very well to the parallel GPU architecture. These techniques would result in malware that is much harder for the average user to detect.

5.1.4 Password Cracking and File Decryption

One area where GPU computing has been leveraged since it first became available is the brute force cracking of password hashes. [1] demonstrates how GPUs are able to generate hashes over 100 times more quickly than CPUs resulting in decreased password cracking times. This capability could be used to make malware much more effective at gaining access to sensitive information and escalated priviledges. Host code could easily access hashes of any user passwords and then crack them in the background on the GPU. Not only would this greatly reduce the time needed for cracking but it would also prevent the increased CPU usage that often reveals the presence of malware. This same technique could also be used to decrypt files on a victim's computer.

5.1.5 Botnet Services

Botnets are at the heart of computer security. They are behind most DoS attacks, most malware distribution, and most spam email. A wide variety of techniques are used by botnet operators to make them more effective and harder to detect, and many of these techniques could benefit from GPU computing. One technique often used to increase the robustness of a botnet is the dynamic calculation of command and control servers. These calculations are typically done on the bot's CPU, but they could be accelerated by using the GPU. This would allow more advanced schemes to be used making them harder to predict. In a similar fashion, the power of a GPU could be used to encrypt the communications between a bot and the command and control server. A more complex encryption scheme could be used to increase security, and the host would be available to do other tasks. Distributed GPU computing could also be leveraged to do things like distributed password cracking or bitcoin mining.

5.2 Suggested Mitigations

The wide variety of possible malicious applications described will likely require a wide variety of mitigations to properly combat. It's likely that anti-malware products will have to advance to the point that analysis of GPU binary files is possible. This will likely require more disclosure from Nvidia, since the doumentation is sparse and the disassembler is not very robust. [33] is an open source project that attempts to support reverse engineering this type of file, but without any official assistance from Nvidia it is likely not robust enough to be relied on by security vendors. New encryption schemes will also need to be developed that take into consideration the unique performance characteristics of modern GPUs. They could be specifically designed to prevent GPU password cracking.

CHAPTER 6. CONCLUSION

6.1 Contributions

This work began with a discussion of the last decade of GPU computing and the current state of GPU security research. It was then demonstrated how vulnerable the current GPU security landscape is. This was done by developing two main attacks. The first one had multiple varieties, but the focus was on the DoS vulnerability inherent to the way that modern GPUs communicate with CPUs. This attack was tested and found to be effective. Based on the specifics of the individual machine, the results ranged from several seconds of unresponsiveness followed by a graphics driver reset, to a complete system crash. The second main attack focused on exploiting the lack of memory protections on GPUs to leak information from one program to another. The attack was effective, and the widespread possibilities of this type of attack were discussed.

We not only demonstrated these vulnerabilities through working attacks, but we also discussed the implications of these attacks and how they could be leveraged to create serious security breaches. While not comprehensive, this work also included discussion of various mitigation strategies to defend against each of the suggested vulnerabilities. Examples of existing mitigations were also mentioned, but most of these were found to be inadequate.

Our work concluded with a survey of possible ways that a GPU could be used to assist malware and make it more effective and harder to detect. Our investigation of some suggestions resulted in the conclusion that they were not possible at the moment, but others were very possible. Each of these presents a very real threat, and it's possible that some are already being utilized by attackers. Suggested mitigations were also mentioned to deal with several of the possible GPU-assisted malware schemes.

6.2 Future Work

This research is meant to be a first step towards improving the GPU security landscape. As such, it leaves the door wide open for future work. The ideas presented here could certainly be expanded upon, and these vulnerabilies are definitely not comprehensive in scope. The field of GPU security is in its infancy, and it is almost certainly going to grow quickly in the near future.

There are several options for improving the basic DoS attacks presented earlier. As mentioned, code could be written to query the device and determine what the best parameters would be for each specific system. Additionally, some extra logic could be added to freeze the GPU for a short enough time that the OS's timer does not kick in and reset the driver, and then keep freezing it repeatedly. This would result in a near total system crash even on those OSs that have timers. The DoS attacks could also be modified to utilize complex operations rather than actual infinite loops. If created correctly, these could achieve the same DoS while being protected from future mitigations that look for infinite loops.

Chapter 4 mentioned a few ways that the information leakage attack could be extended, and there are many more possibilities. This vulnerability leaves so many applications so wide open to attack. The basic CUDA code could easily be extended to recover larger chunks of data and do some analysis on the recovered data. This could be tailored to attack anything from a website to a video game to a scientific computation. It's possible that it could even be used to steal the contents of a user's desktop at any given time.

Chapter 5 discusses several ways in which GPUs could be used to assist malware. None of these have been extensively studied, and each one would make an interesting topic for future research. Malware authors are likely already working on ways to leverage the GPU to gain an advantage, so it would be wise for academic study to be done in this area as well. Possible attacks for each of the methods listed could be developed, and working implementations of the various mitigations would greatly further this field.

BIBLIOGRAPHY

[1] Hashcat advanced password recovery. http://hashcat.net/oclhashcat-plus/. Accessed: 03/26/2013.

[2] Kickjs. http://www.kickjs.org/example/shader_editor/shader_editor.html. Accessed: 03/30/2013.

[3] Insight 64. Amd fusion family of apus: Enabling a superior, immersive pc experience. http://www.amd.com/us/Documents/48423_fusion_whitepaper_WEB.pdf. Accessed: 04/10/2013.

[4] Mark Aiken, Manuel Fähndrich, Chris Hawblitzel, Galen Hunt, and James Larus. Deconstructing process isolation. In *Proceedings of the 2006 workshop on Memory system performance and correctness*, MSPC '06, pages 1–10, New York, NY, USA, 2006. ACM.

[5] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, October 2009.

[6] Alberto Benin, G. Riccardo Leone, and Piero Cosi. A 3d talking head for mobile devices based on unofficial ios webgl support. In *Proceedings of the 17th International Conference on 3D Web Technology*, Web3D '12, pages 117–120, New York, NY, USA, 2012. ACM.

[7] M. Bishop, S. Engle, D. Howard, and S. Whalen. A taxonomy of buffer overflow characteristics. *Dependable and Secure Computing, IEEE Transactions on*, 9(3):305–317, 2012.

[8] M.A. Bochicchio, A. Longo, and L. Vaira. Extending web applications with 3d features. In *Web Systems Evolution (WSE), 2011 13th IEEE International Symposium on*, pages 93–96, 2011.

[9] Andr R. Brodtkorb, Trond R. Hagen, and Martin L. Stra. Graphics processing unit (gpu) programming strategies and trends in gpu computing. *Journal of Parallel and Distributed Computing*, 73(1):4 – 13, 2013. ¡ce:title¿Metaheuristics on GPUs¡/ce:title¿.

[10] Herwin Chan, Patrick Schaumont, and Ingrid Verbauwhede. Process isolation for reconfigurable hardware. In *2006 International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA06) (Distinguished Paper*, pages 164–170, 2006.

[11] J.Y. Chen. Gpu technology trends and future requirements. In *Electron Devices Meeting (IEDM), 2009 IEEE International*, pages 1–6, 2009.

[12] Loıc Duflot, Yves-Alexis Perez, Guillaume Valadon, and Olivier Levillain. Can you still trust your network card. *CanSecWest/core10*, pages 24–26, 2010.

[13] Paul Ferguson. Network ingress filtering: Defeating denial of service attacks which employ ip source address spoofing. 2000.

[14] David Geer. Taking the graphics processor beyond graphics. *Computer*, 38(9):14–16, 2005.

[15] John Douglas Howard. *An analysis of security incidents on the Internet 1989-1995*. PhD thesis, Pittsburgh, PA, USA, 1998. UMI Order No. GAX98-02539.

[16] Google Inc. Angle project. https://code.google.com/p/angleproject/. Accessed: 03/21/2013.

[17] Google Inc. Gpu accelerated compositing in chrome. http://www.chromium.org/developers/design-documents/gpu-accelerated-compositing-in-chrome. Accessed: 03/25/2013.

[18] The Khronos Group Inc. The opengl es shading language. http://www.khronos.org/files/opengles_shading_language.pdf. Accessed: 03/21/2013.

[19] V. Kindratenko and P. Trancoso. Trends in high-performance computing. *Computing in Science Engineering*, 13(3):92–95, 2011.

[20] D. Luebke. Cuda: Scalable parallel programming for high-performance scientific computing. In *Biomedical Imaging: From Nano to Macro, 2008. ISBI 2008. 5th IEEE International Symposium on*, pages 836–838, 2008.

[21] S.A. Manavski. Cuda compatible gpu as an efficient hardware accelerator for aes cryptography. In *Signal Processing and Communications, 2007. ICSPC 2007. IEEE International Conference on*, pages 65–68, 2007.

[22] Microsoft. Tdr registry keys. http://msdn.microsoft.com/en-us/library/windows/hardware/ff569918(v=vs.85).aspx. Accessed: 03/20/2013.

[23] G.E. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, 1998.

[24] Tim Newsham. Format string attacks, 2000.

[25] J. Nickolls and W.J. Dally. The gpu computing era. *Micro, IEEE*, 30(2):56–69, 2010.

[26] Gilles Pags and Benedikt Wilbertz. Gpgpus in computational finance: massive parallel computing for american style options. *Concurrency and Computation: Practice and Experience*, 24(8):837–848, 2012.

[27] R.R. Rejimol Robinson and C. Thomas. Evaluation of mitigation methods for distributed denial of service attacks. In *Industrial Electronics and Applications (ICIEA), 2012 7th IEEE Conference on*, pages 713–718, 2012.

[28] Daniel Reynaud. GPU powered malware. In *Ruxcon*, Sydney Australie, 11 2008.

[29] C.L. Schuba, I.V. Krsul, M.G. Kuhn, E.H. Spafford, A. Sundaram, and D. Zamboni. Analysis of a denial of service attack on tcp. In *Security and Privacy, 1997. Proceedings., 1997 IEEE Symposium on*, pages 208–223, 1997.

[30] Peter Schwabe. Graphics processing units. In Kostas Markantonakis, editor, *Secure Smart Embedded Devices: Platforms and Applications*. Springer-Verlag, 2013 (to appear). to appear, http://cryptojedi.org/papers/#gpus.

[31] Patrick Stewin and Iurii Bystrov. Understanding dma malware. In Ulrich Flegel, Evangelos Markatos, and William Robertson, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, volume 7591 of *Lecture Notes in Computer Science*, pages 21–41. Springer Berlin Heidelberg, 2013.

[32] PaX Team. Original design & implementation of pageexec, 2000.

[33] Wladimir J. van der Laan. Cubin utilities. https://github.com/laanwj/decuda/wiki. Accessed: 03/26/2013.

[34] Victor van der Veen, Nitish dutt Sharma, Lorenzo Cavallaro, and Herbert Bos. Memory errors: the past, the present, and the future. In *Proceedings of the 15th international conference on Research in Attacks, Intrusions, and Defenses*, RAID'12, pages 86–106, Berlin, Heidelberg, 2012. Springer-Verlag.

[35] 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.

[36] C.J. Veenman. Statistical disk cluster classification for file carving. In *Information Assurance and Security, 2007. IAS 2007. Third International Symposium on*, pages 393–398, 2007.

[37] S. D. Walsh, J. Myre, M. O. Saar, and D. Lilja. Multi-GPU, Multi-core, Multi-phase Lattice-Boltzmann Simulations of Fluid Flow for the Geosciences. *AGU Fall Meeting Abstracts*, page D983, December 2009.

[38] A. Wood and J.A. Stankovic. Denial of service in sensor networks. *Computer*, 35(10):54–62, 2002.