5-2022

Analysis of GPU Memory Vulnerabilities

Jarrett Hoover

Analysis of GPU Memory Vulnerabilities

An Undergraduate Honors College Thesis

in the

Department of Computer Science and Computer Engineering
College of Engineering
University of Arkansas
Fayetteville, AR
April 2022

by

Jarrett Hoover

Analysis of GPU Memory Vulnerabilities

Jarrett D. Hoover
Department of Computer Science and Computer Engineering
University of Arkaansas
Fayetteville, AR, 72701, USA
jdh074@uark.edu

Dale R. Thompson
Department of Computer Science and Computer Engineering
University of Arkansas
Fayetteville, AR, 72701, USA
drt@uark.edu

***Abstract*—Graphics processing units (GPUs) have become a widely used technology for various purposes. While their intended use is accelerating graphics rendering, their parallel computing capabilities have expanded their use into other areas. They are used in computer gaming, deep learning for artificial intelligence and mining cryptocurrencies. Their rise in popularity led to research involving several security aspects, including this paper's focus, memory vulnerabilities. Research documented many vulnerabilities, including GPUs not implementing address space layout randomization, not zeroing out memory after deallocation, and not initializing newly allocated memory. These vulnerabilities can lead to a victim's sensitive data being leaked to an attacker, an impactful threat considering the usages of GPU computing presented. In this paper, we attempt to implement these vulnerabilities on an NVIDIA GPU to determine if any advancements in memory architecture have been made since the documentation of such vulnerabilities. This work demonstrates that the lack of attention to security in early GPU development has been adjusted to appropriately match a level of concern for a computing component that numerous industries rely on.**

Keywords—Graphics Processing Unit (GPU), Address space layout randomization (ASLR), Allocation/Deallocation, Global memory, Memory leakage, Host, Device

I. Introduction

Graphics Processing Units (GPUs) have come a long way since their inception. While NVIDIA popularized the term "GPU" in 1999, their invention came long before that. NVIDIA's original definition of the GPU was a single chip processor with integrated transform, lighting, triangle setup/clipping, and rendering engines capable of processing a minimum of 10 million polygons per second [14]. Since this definition describing their accelerated graphics rendering capabilities, they have had an extreme increase in popularity and function. Today they are used for mining cryptocurrency, training neural networks for artificial intelligence, and even common web browsers use GPUs rendering capabilities when available [10, 11]. Browsers such as Chrome use GPUs for video encoding and playback acceleration to lighten the load on CPUs [1]. This growth in popularity can be attributed to these parallel computing devices being fully programmable with a high throughput, efficiency, and ability to offload CPU work [3]. However, with all of their advantages, security was not investigated enough in early development, and they were thought of as the weakest link in the security chain. Any problem with GPUs can also affect all GPU accelerated applications, and today the list continues to grow. The original data they dealt with may not be considered sensitive, but it certainly is today. Many problems discovered early on had to do with their memory architecture. Threats on GPU memory were overlooked at first, and it was discovered that sensitive data could be extracted from memory residues [3]. This extraction process largely dealt with GPUs not initializing newly allocated memory [2]. In this work, we focus on these documented vulnerabilities in GPU memory, specifically global memory.

The contributions of this work are: (1) proving that 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 now initialized to zero.

The following section discusses the related work of GPU memory vulnerabilities and how these vulnerabilities can be used to extract various forms of sensitive data. The presented related work was used heavily to test vulnerabilities in near exact ways and then expanded and altered in this work. Section III describes the characteristics of early GPU memory architecture that permitted these vulnerabilities. In Section IV, we highlight key functions in CUDA that deal with memory management. Section V introduces the theories behind the attacks on global memory. The evaluation of these attacks is presented in Section VI, followed by our conclusions in Section VII.

II. Related Work

Many researchers noticed memory leakage security issues and exploited them in different ways. The authors in [1] first noticed the lack of address space layout randomization (ASLR), a policy that aids in process isolation which is impactful to computer security. They address that CPUs implement process isolation with ASLR and virtual memory, and they demonstrate that GPUs implemented neither at the time. Through GPU programming with CUDA, they prove that same memory allocations result in the exact same address in different program executions. Additionally, they present a proof-of-concept attack that steals information between CUDA programs due to this lack of ASLR paired with memory not being zeroed out prior to allocation or after deallocation.

Similarly, the authors in [2] disclosed sensitive data remaining in GPU memory through end of context and end of

kernel attacks. They revealed uninitialized memory problems through these attacks that read all global memory right after a victim finishes execution on a shared GPU. In other words, they successfully obtained data released after the destruction of a victim programs GPU context. Their attacks could retrieve results of kernel computations such as decrypted plaintext and rendered images. Other data forms released include kernel code, constant data, and call-by-value arguments of kernels. Furthermore, they exhibited an ability to infer web browsing history of victims using pixel analysis on the recovered data.

Taking a different approach than memory leakage, the authors in [3] discovered that the execution time of a GPU kernel is linearly proportional to the number of unique cache line requests generated during kernel execution on GPU architectures. This architectural flaw allowed for a timing attack on GPUs. They used this property to extract AES encryption keys when cryptography is implemented by GPUs.

Another example of memory leakage is presented by the authors in [4]. They demonstrated direct sensitive information recovery from real-world applications including Chrome and Adobe due to memory management vulnerabilities. They found GPU memory inference attacks to be much more serious than researchers had previously thought. They explained that GPU memory is managed by the GPU itself and thus may violate some security policies that are normally enforced by the operating system and CPU. Like others, they also documented that newly allocated memory is not cleared to zero and that the GPU does not erase memory data before the released memory space gets reallocated. They performed a similar attack of filling global memory with a predefined value, monitoring available memory, and performing a memory dump to analyze what had been left over from the victim. Doing this, they were able to obtain usernames, credentials, and credit card numbers. They furthered their attack to recover victims' data after they viewed webpages or PDFs and implemented image analysis on the recovered data. Their attack was shown to work on Chrome and Adobe as previously mentioned, but also GIMP and MATLAB.

As the early 2010s progressed and papers such as these continued to be published, the authors in [5] published a survey of techniques for improving GPU security. Several of the aforementioned papers are cited in this publication as well. New vulnerabilities mentioned include the ability to perform a register spilling attack. Such an attack is performed by reserving more registers than available and the overflow goes to global memory, allowing an attacker to access this data even though it is not registered to them and has not been deallocated. This publication features no work of their own but have nearly 50 sources of GPU security issues and an entire section on the focus of this work, memory leakages.

The most recent paper we present in related work comes from the authors in [6]. They illustrate even more memory concerns including GPUs lacking prevention of threads of a kernel to access the contents stored in local and private memories written by threads of other kernels. Written in 2020, they cite that when multiple users share a GPU, information leakage can still occur between concurrently running processes or from recently terminated process because of allocation issues, despite their documentation for at least 7 years. This dissertation goes as in depth as providing mitigations to some of these vulnerabilities.

Despite all this previously published work on the same memory leakage vulnerabilities, they persisted for many years. We believe these issues still deserve attention, as we are unaware of any work proving their elimination.

III. Early Memory Architecture

To gain a full understanding of the content of this work, a brief discussion of the characteristics of early architecture of GPU memory is beneficial. GPUs had little caching but a large and fast RAM system to feed data into the many cores. Additionally, they did not initially implement virtual memory, a policy to aid in process isolation [1]. They also featured no preemptive scheduling, allowing for the previously mentioned end of kernel attack. GPUs contain a large amount of independent memory as well [2]. GPUs also contain integrated off-chip DRAM memory called device memory. CPUs transfer data to and from this device memory before and after they launch the kernel. The global memory we focus on is found in this device memory. Another component of GPUs, streaming multiprocessors, of which there are many, all share this global memory [13]. Compared to CPUs, cache sizes are much smaller but have much higher bandwidth [3]. GPUs also have memory space dedicated to them and it is used and managed exclusively by the GPU, explaining the need for data to be transferred to and from host to device [4]. Finally, all streaming multiprocessors share an L2 cache while they each have their own L1 caches which are write-back by default [6].

IV. CUDA Memory Management

The vulnerabilities that this work assessed were performing using an application called CUDA. This toolkit is a parallel computing platform and programming model created by NVIDIA to interact with their GPUs [15]. It has allowed users to perform research and build GPU-accelerated applications for many years, and this work takes advantage of this platform's flexibility. We spent a large amount of time learning and understanding CUDA memory management through documentation, so we find that a description of some common syntax and functions to be valuable.

The primary syntax rule that permitted memory management on the GPUs specific memory was the __global__ declaration. This declaration before a function signals the CUDA compiler that you are declaring kernel [8]. This means that the function is called on the host (CPU) but executed on the device (GPU). In these functions, normal C-like variables such as arrays indicated reading or writing from the GPU global memory, which were needed features for our purposes. In these kernel functions, you could simulate victim activities or perform attacker actions depending on your purpose.

Many functions found in the CUDA Toolkit documentation were highly used as well. The most impactful function was cudaMalloc. This function allocated memory on the device. Its parameters were a predefined pointer that the function would return the allocated memory location and the size of your requested allocation in bytes. It is trivial to understand this function, but it was used in several different contexts. This could be a victim storing their sensitive information or an attacker

allocating the same memory to see what raw data remained. Knowing the memory size of the vulnerable GPU would also allow you to allocate the entirety of global memory due to the size parameter.

The function of next importance was cudaMemcpy. This function copies data between the host and the device. Its parameters were the destination memory address, the source memory address, the size in bytes to copy, and a CUDA specific identifier to indicate the type of transfer. The types of transfer include host to device, device to host, device to device, or even host to host. This work primarily used host to device and vice versa. The memory regions requested must be registered with CUDA so you cannot try and access deallocated memory. Our purposes for this function included writing to global memory with host to device transfers and reading from global memory with device to host transfers.

Another trivial function, the complement to cudaMalloc, is cudaFree. This function simply frees the memory on the device. Its only parameter is the device pointer to the memory to be freed. If the pointer was invalid or to unregistered memory, a cudaError variable with a value of invalid value was returned referring to the pointer. Similarly, a successful deallocation of memory returned a cudaError variable with a value of success. Many of these memory management functions returned this cudaError variables and error checking functions could be written easily if the desire or need was there. This simple function was used critically because in order to test how the GPU handled deallocation, this function had to be called properly.

A much less frequently used function was cudaMemset. This function initializes or sets device memory to a specified value. Its parameters are a pointer to device memory, a value to set for each byte of specified memory, and a size in bytes to set. This function is perfect for filling global memory with a value known only to you, but knowledge of the exact memory size of your device is required, which is easy to find. Use in this context allows for quick analysis of a global memory dump to find victims' data.

The final function of importance was cudaMemGetInfo. This function was crucial for attack algorithms as it returns the amount of free and total device memory. The parameters are two variables that indicate the returned free memory in bytes, and the returned total memory in bytes. The returned memory attributes are of the current context of the GPU. This function can be used to monitor memory by constant polling to detect if a victim has used memory. When the amount of free memory changes, you can infer whether victims have allocated or deallocated memory. There are many other functions found in the CUDA documentation for memory management, but these were of high significance to the work performed [7].

V. Attacks On Global Memory

In this section, we cover the theories and concepts behind the attacks implemented in this work. They can be categorized based on whether the attack is dependent on ASLR or not. There are six attacks that attempt to exploit lack of ASLR, or lack of zeroed out memory by allocation/deallocation, from many different angles. Some are taken from the related work section to see if they are still applicable, and some are adaptations of those attacks or similar to them.

A. ASLR Dependent

The first attack presented in this work attempts to show the lack of ASLR because of its proven absence in previous work [1]. Address space layout randomization is a computer security technique that randomizes the location where system executables are loaded into memory. It makes buffer overflow attacks difficult because they involve the attacker knowing the location of an executable in memory [12]. A simple program can show the lack or presence of ASLR. Code that shows the exact same memory allocations will result in different addresses during different program executions demonstrates the function of ASLR. The code we used to prove or disprove ASLR is acquired from the authors in [1] due to its simplicity yet undeniable results.

The next attack theory exploited the lack of ASLR paired with memory not being cleared after deallocation or with allocation. A simple proof of concept attack also from the authors in [1] was used to fulfill this theory. This attack involved two CUDA programs where one leaks information from memory to the other. In this scenario, the first program is the victim, who saves/writes information to memory. The second program, the attacker, allocates new memory, which from the lack of ASLR would be the same location, and retrieves the information that the victim wrote to memory. If there is no implementation of ASLR or clearing of memory, this attack should work just as it did for the creators. This attack is small and simple, but if it is successful the vulnerability can easily be utilized in much more dangerous and destructive attacks.

Another attack exploits the scenario if the victim forgets to deallocate their memory. If victims forget to free their sensitive information, would it be possible to expose a memory leakage as others have done as seen in the related work section? A simple attack can be created by modifying the proof-of-concept attack mentioned earlier. This attack still features two CUDA programs, a victim and attacker. However, in this scenario the victim does not deallocate their memory usage. If this attack works, there is proof that newly allocated memory is not initialized. This is important because it is often hard to determine whether deallocation, allocation, or even program termination is the cause of zeroed out memory. Once again, this attack is dependent on the enforcement of ASLR.

The last attack that depends on the policy of ASLR attempts to eliminate a previously mentioned issue, whether memory is cleared after program termination, regardless of the freeing of memory. This attack combines all the previously used code into the same program, creating some new possibilities due to the victim and attacker working under the same program execution. In this attack, we try to access the same memory right after it has been deallocated to see if it remains. In this setting, the unknown way a GPU handles memory after program termination is eliminated. Simple changes to the code can also test a few other angles. The victim code could not free their memory, but this would only be fruitful if ASLR is not in play. A final thought on this attack was removing the second allocation statement, which would be performed by the attacker. However, this will only prove successful if the GPU allows access to memory that has

been deallocated in the same program, which is not a concept we concentrated on. Essentially, the attacker would be trying to access non-registered memory using a pointer from the victim's allocation statement, which CUDA documentation states should return an error.

B. ASLR Independent

The next two attack scenarios will work independent of ASLR enforcement and focus primarily on data being left over in global memory after deallocation. The first attack we present was creating using a template provided in the CUDA toolkit samples. This example program takes an array on the host, fills it with values increasing sequentially, transfers this array to global memory on the GPU, performs some multithreaded calculations on the GPU, and ends by transferring those results back to the host. This template project was perfect to modify because it dealt with memory allocation and deallocation using GPU global memory, all pillars of this research. Another reason was that this was a proper CUDA project, written by the creators themselves, so any attempts to exploit vulnerabilities were sure to be official. The first addition we made was adding a kernel function that takes in a pointer to the GPU array and tries to steal values after its deallocation. When we call this function, we use a captured pointer value before the array's deallocation. This function attempts to bypass some of features of previously used CUDA functions that do not allow for access to unregistered memory. This is because it appears to access array values just like a non CUDA program would, but its kernel declaration means it runs on the GPU. In this same project we also attempt a non-kernel function approach using the same captured pointer, but this approach does depend on the most likely secure CUDA memory management functions.

The final attack program is a modification of the end of context attack published by the authors in [2]. The code for the attack was also published [9]. In their work, they use this attack to steal victims web browsing data. This attack essentially fills global memory with a value known only to the attacker, deallocates that memory, and waits for a victim to come use GPU global memory. They do this by monitoring the amount of free memory using CUDA memory management functions previously discussed. As soon as the victim programs context is terminated, they perform a memory dump of all global memory. This strategy eliminates the issues of unregistered memory because all global memory is accessed. When analyzing the memory dump, any data that does not match the prefilled value is considered a victim's sensitive data. Some small changes we made in our implementation involve memory being filled with 1s, changed memory specifics to match our GPU, and the victims were CUDA toolkit examples that access device memory. If memory is not zeroed out as cited many times, this attack should work as it did in this related work. However, if results showed that memory was zeroed out, you could change the memory dump analysis code to only check for non-zero data. If there was none, you could prove that this vulnerability no longer remains. Another feature added to this attack was compiling it with compilation flags that disabled the L1 caches of the GPU [16]. This was done to eliminate the possibility that data was never reaching global memory. Additionally, a separate victim program was compiled with the same flags and had unique victim data to ensure the data was making it to global memory. This data was 2 GB allocations and was initialized based on a unique key array like the way the authors in [2] filled global memory. The same method was applied to the victim data instead of prefilled values.

VI. Evaluation

A. Experimental Setup

In this section we show the experimental results of the experiments we performed to evaluate our proposed programs. The evaluation setup consisted of a NVIDA Quadro P4000 GPU running on an Ubuntu operating system. This GPU had 8 GB of memory. The CUDA toolkit was downloaded on this workstation according to installation instructions, ensuring for proper compiling and running of code. The code was compiled and ran in the terminal in a directory created in the toolkit samples directory to confirm all needed libraries were present. Terminal compiling was done using the nvcc command in all cases except for the program that used the template project. This provided project folder contained its own make file to run for compiling.

B. Results

The experiments consisted of six programs attempting to exploit memory vulnerabilities. The first program, intended to prove the absence of ASLR, resulted in the finding that the setup did implement ASLR, as seen in Fig. 1 where different program executions resulted in different memory locations. However, the other programs that depended on ASLR were still ran to see if pairing this documented vulnerability with other vulnerabilities such as memory leakage would result in a successful attack.

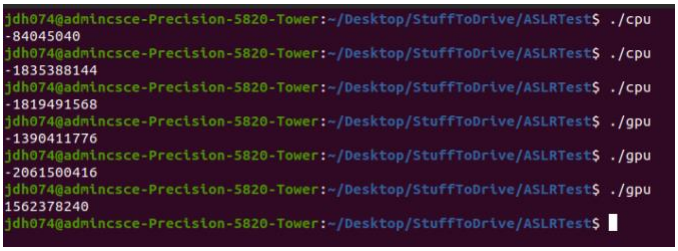

Fig. 1. ASLR implementation on CPU and GPU.

The second attack theory tried to exploit the uncleared memory vulnerability along with a lack of ASLR. We found that no memory leakage occurred and that only the value 0 could be recovered, indicating memory was zeroed out either after deallocation or along with new allocations. An example of this can be found in Fig. 2.

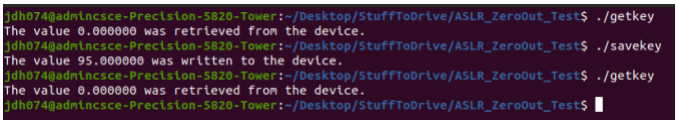

Fig. 2. Value of 0 recovered prior to and after writing to memory.

The next program, where a victim does not deallocate their memory, ended with identical results, indicating that either memory is cleared after program termination or with new allocations. These findings are in Fig. 3.

Fig. 3. Value of 0 recovered with no deallocation of memory.

The final program to cover all scenarios of ASLR implementation and memory clearing vulnerabilities, involved work in the same program execution. The first test has a victim writing to memory, deallocating it, and an attacker attempts to retrieve that data immediately. We found that similarly, only values of 0 could be retrieved, as seen in Fig. 4. The second test with this program has a victim forgetting to deallocate their memory by simply commenting out the freeing of memory statement, but once again, the same results occurred. These can be found in Fig. 5. The final test with this program had the attacker not using a new allocation statement but instead using the victim's pointer to the deallocated memory. Here, if a default value was provided in the code, that was the value that was retrieved, and in our test the arbitrary value of 2.5 was default. This was because attempting to read from unregistered memory returns an error, and the default value never actually changes. These results can be seen in Fig. 6. All attempts to exploit the documented vulnerabilities were unsuccessful in these discussed attempts. However, they all have the possibility of being thwarted due to ASLR. For this reason, we carried out the final two attacks that ASLR cannot affect.

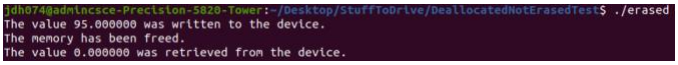

Fig. 4. Value of 0 recovered in same program execution.

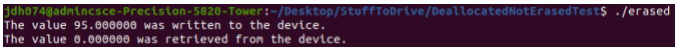

Fig. 5. Value of 0 recovered with no deallocation of memory.

Fig. 6. Default value of 2.5 retrieved.

The next two attacks, although independent of ASLR, showed similar results. The attack using the template CUDA project was unable to recover any values after deallocation. We found this to be because the GPU policies implemented prevented any attempts to access deallocated memory, even in the kernel function that was independent of CUDA memory management functions. The error thrown was an invalid value error in the cudaMemcpy function. The end of context attack, if ran exactly as the original authors in [2] coded it, would print the entire global memory which was zeros. This was because they printed any values different from the prefilled values, which were not zero. This proved that memory was cleared out sometime between deallocation and allocation of the entire global memory. These results can be seen in Fig. 7. The alteration of the attack exhibited the same findings. We changed the program to print any data that was not zero, to see if any victim data could be found. This resulted in nothing from the memory dump being printed out, indicating the entirety of the global memory that we recently allocated were zeros. These findings are found in Fig. 8. The same results occurred in the extra runs when executing this attack with L1 caches disabled for the attacker and the victim. No victim data was able to be recovered, even when using data larger than the L2 cache size of 2 MB that was sure to be found in global memory, not the cache.

Fig. 7. Global memory full of 0s printed out.

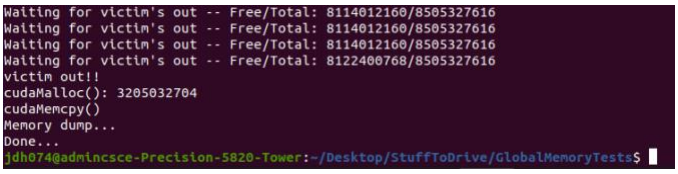

Fig. 8. No data printed due to zeroing out.

C. *Issues*

The primary issue we faced in this work was that the reason for the failure to exploit a certain vulnerability was sometimes unclear. In other words, there are multiple reasons for a failed attack and sometimes it was impossible to determine the exact reason. Therefore, we tried to take every approach angle and focused on only three specific vulnerabilities. For some of the attacks that relied on ASLR, it would be unclear whether memory leakage was not occurring due to proper memory zeroing, or the implementation of ASLR. Other reasons for failure could include data being cached and not actually making it to global memory, a fact not heavily considered in this work. Another issue was the lack of documentation by NVIDIA to find if policies have been implemented to fix these vulnerabilities and when.

VII. Conclusion

In this work, several approaches were taken to exploit the GPU vulnerabilities of no address space layout randomization implementation, and no zeroing of global memory after deallocation or with new allocations. However, all these previously documented attacks appear to be fixed. The work demonstrated this through six different CUDA programs which were a compilation of related work code, modifications to this code, and a new program using a NVIDIA provided template project. The documented security issues with global memory up to 2020 appear to have been successfully mitigated, a relieving statement given the substantial use of GPUs today. It is worth noting that caches were taken into consideration to the best of our ability, but there is a possibility that some victim

data would be cached in L2 which cannot be disabled like L1, leading to attack failures because the data never makes it to global memory, where these vulnerabilities were documented.

REFERENCES

[1] Patterson, Michael, "Vulnerability analysis of GPU computing" (2013). *Graduate Theses and Dissertations*. 13115. https://lib.dr.iastate.edu/etd/13115

[2] S. Lee, Y. Kim, J. Kim and J. Kim, "Stealing Webpages Rendered on Your Browser by Exploiting GPU Vulnerabilities," *2014 IEEE Symposium on Security and Privacy*, 2014, pp. 19-33, doi: 10.1109/SP.2014.9.

[3] Z. H. Jiang, Y. Fei and D. Kaeli, "A complete key recovery timing attack on a GPU," *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 394-405, doi: 10.1109/HPCA.2016.7446081.

[4] Z. Zhou, W. Diao, X. Liu, Z. Li, K. Zhang and R. Liu, "Vulnerable GPU Memory Management: Towards Recovering Raw Data from GPU," *Proceedings on Privacy Enhancing Technologies*, 2017, pp. 57-73, doi: 10.1515/POPETS.2017.0016.

[5] S. Mittal, S. B. Abhinaya, M. Reddy and I. Ali, "A Survey of Techniques for Improving Security of GPUs," *Hardware and Security Systems Journal 2*, 2018, pp. 266-285, doi: 10.1007/s41635-018-0039-0.

[6] Naghibijouybari, H. (2020). Security of Graphics Processing Units (GPUs) in Heterogeneous Systems. *UC Riverside*. ProQuest ID: Naghibijouybari_ucr_0032D_14214. Merritt ID: ark:/13030/m5h475c2. Retrieved from https://escholarship.org/uc/item/6jx346hq

[7] CUDA Tookit Documentation v11.6.2. https://docs.nvidia.com/cuda/index.html (accessed Nov. 16, 2021).

[8] CUDA syntax. http://www.icl.utk.edu/~mgates3/docs/cuda.html (accessed Dec. 9, 2021).

[9] Sangho Lee Github Code. https://github.com/sangho2/gpu-uninit-mem/blob/master/attack.cu (accessed Feb. 15, 2022)

[10] A. Kordek. "Uses for GPUs: 4 Reasons Other Than Gaming." Inmotion Hosting. https://www.inmotionhosting.com/support/product-guides/private-cloud/additional-resources/uses-for-gpus-other-than-gaming/ (accessed Mar. 3, 2022).

[11] H. Klein. "Impact of GPU Acceleration on Browser CPU Usage." Helge Klein. https://helgeklein.com/blog/impact-gpu-acceleration-browser-cpu-usage/#:~:text=Chrome%3A%20GPU%20Usage&text=Obviously%2C%20Chrome%20uses%20the%20GPU,but%20also%20for%202D%20rendering.&text=Especially%20during%20video%20playback%2C%20but,GPU%20is%20still%20used%20extensively (accessed Mar. 7, 2022).

[12] S. Shea. "address space layout randomization (ASLR)." TechTarget. https://www.techtarget.com/searchsecurity/definition/address-space-layout-randomization-ASLR (accessed Feb. 24, 2022).

[13] A. Priyadarshana. "CUDA – GPU Memory Architecture." Medium. https://ashanpriyadarshana.medium.com/cuda-gpu-memory-architecture-8c3ac644bd64 (accessed Mar. 1, 2022).

[14] "History of GPUs." XOTIC PC. https://xoticpc.com/blogs/news/history-of-gpus#:~:text=Back%20in%201999%2C%20NVIDIA%20popularized,card%20to%20rule%20them%20all (accessed Mar. 2, 2022).

[15] F. Oh. "What is CUDA?" NVIDIA. https://blogs.nvidia.com/blog/2012/09/10/what-is-cuda-2/ (accessed Feb. 22, 2022).

[16] CUDA Toolkit Documentation. "4.2.9.1. Ptxas Options." NVIDIA. https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#ptxas-options (accessed Apr. 19, 2022).