GPU Memory Exploitation for Fun and Profit

Yanan Guo, *University of Rochester;* Zhenkai Zhang, *Clemson University;*
Jun Yang, *University of Pittsburgh*

https://www.usenix.org/conference/usenixsecurity24/presentation/guo-yanan

This paper is included in the Proceedings of the
33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

Open access to the Proceedings of the
33rd USENIX Security Symposium
is sponsored by USENIX.

GPU Memory Exploitation for Fun and Profit

Yanan Guo*†
University of Rochester

Zhenkai Zhang*
Clemson University

Jun Yang
University of Pittsburgh

Abstract

As modern applications increasingly rely on GPUs to accelerate the computation, it has become very critical to study and understand the security implications of GPUs. In this work, we conduct a thorough examination of buffer overflows on modern GPUs. Specifically, we demonstrate that, due to GPU's unique memory system, GPU programs suffer from different and more complex buffer overflow vulnerabilities compared to CPU programs, contradicting the conclusions of prior studies. In addition, despite the critical role GPUs play in modern computing, GPU systems are missing essential memory protection mechanisms. Consequently, when buffer overflow vulnerabilities are exploited by an attacker, they can lead to both code injection attacks and code reuse attacks, including return-oriented programming (ROP). Our results show that these attacks pose a significant security risk to modern GPU applications.

1 Introduction

Graphic Processing Units (GPUs) were originally designed and used for high-quality graphics rendering. However, over the past decade, they have evolved into general-purpose computing platforms. Due to their high-throughput capabilities, GPUs are now used in various fields, including weather prediction [37], crypto-currency mining [28], and bioinformatics analysis [34]. In addition, today GPUs have become the *de facto* standard choice for running deep learning applications [9, 17, 24, 29, 30, 47, 52, 55, 56]. Given the growing significance of GPUs, NVIDIA recently announced the Grace Hopper Superchip [42] which is designed for giant-scale artificial intelligence (AI) and high-performance computing (HPC) applications. This superchip combines the NVIDIA Grace CPUs and Hopper GPUs using the high-speed interconnect, NVLink [43]. This wide employment of GPUs inevitably urges a thorough study on their security implications.

Memory safety violations (memory errors) have long been a significant security concern for computing systems. These violations are the most common root cause of modern exploits (attacks). For example, buffer overflows can allow attackers to overwrite return addresses and thus hijack the control flow of a program, potentially leading to the execution of malicious code. In fact, reports from Google and Microsoft show that memory errors account for around 70% of all security issues addressed in their products [23, 39]. Memory safety violations, together with the associated exploitation techniques, have been widely studied for CPU programs. Modern CPUs have even implemented certain built-in defense mechanisms against these vulnerabilities (e.g., [31, 32, 59]). However, the vulnerabilities in GPU programs have not received the same attention.

CUDA [41], developed by NVIDIA, is one of the most popular general-purpose GPU programming languages in use today. Since CUDA is extended from C and C++–languages known for their memory-unsafe characteristics–there is a concern that CUDA programs could have similar memory safety vulnerabilities. In this paper, we delve into this concern, aiming to answer the following questions:

Can memory errors occur in CUDA programs running on NVIDIA GPUs? If so, what types of attacks can arise from these errors?

Several prior studies have explored the memory safety vulnerabilities in CUDA programs [19, 38, 48]. They show that memory safety violations, especially buffer overflows, can occur in CUDA programs as well. However, they also argue that conventional CPU memory exploitation techniques, such as code injection and code reuse, are *inapplicable* for attacking CUDA programs. We found that their investigations have significant limitations.

First, the investigations in these studies are preliminary and lack a comprehensive analysis of the memory safety vulnerabilities inherent to GPU programs. For example, unlike C and C++, CUDA features multiple, distinct memory spaces, aligning with the GPU's specialized memory hierarchy. Data

*These authors contributed equally to this work.
†This work was primarily conducted while the author was affiliated with the University of Pittsburgh.

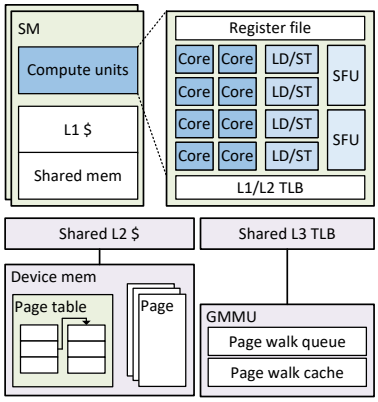

Figure 1: GPU architecture overview.

in different memory spaces have different scopes and are accessed in different manners. Prior studies have only shown that buffer overflows can occur within individual memory spaces. They have not explored whether a buffer overflow in one memory space can directly affect data in another memory space. Second, these studies were conducted on earlier NVIDIA GPUs, with older architectures (Pascal and earlier) and CUDA compute capabilities (sm_60 and earlier). Notably, NVIDIA has made significant changes to the GPU system starting with the Volta architecture (sm_70), which was released in 2017. Thus, the conclusions made in prior studies may not apply to modern GPU architectures.

In this work, we conduct a thorough examination of the buffer overflow problem on modern NVIDIA GPUs. First, we reverse engineer the mechanisms used to access various memory spaces. Specifically, we show how GPU hardware identifies memory references to each memory space and how it conducts address translation for these memory spaces. Based on the reverse engineering results, we demonstrate that an out-of-bounds (OOB) operation on data in one memory space can influence data in another, despite the fact that different memory spaces utilize different instructions for access. Furthermore, we reveal that OOB operations can be exploited to access data beyond their legitimate scopes. For example, one thread can access the local memory belonging to another thread.

Then, we study potential GPU attack methodologies leveraging these buffer overflow vulnerabilities. We found that modern NVIDIA GPUs are missing fundamental memory protection mechanisms. As a result, traditional memory exploitation techniques (which have been mitigated on CPUs) remain feasible on GPUs. For example, GPUs do not distinguish between code and data pages: data pages are executable, and code pages are writable.

In addition, we analyze the mechanics of function calls and returns on modern GPUs. Our investigation reveals that code reuse attacks, such as return-oriented programming (ROP) [50], can be employed against CUDA programs. We further discover that the CUDA driver API library is linked for each CUDA program; certain functions from this library are loaded into GPU memory upon the execution of any CUDA program. Importantly, this library code contains multiple ROP gadgets, including several memory read/write gadgets, which can enable powerful ROP-based attacks.

Finally, we show that the above memory exploitation techniques can be used to attack modern GPU applications such as deep neural network (DNN) inference. For example, by modifying the DNN weights, the attacker can significantly degrade the DNN inference accuracy, reducing it to the same level of random guessing in the most severe cases.

Responsible disclosure. We disclosed our findings to NVIDIA in October 2023, who acknowledged our work and requested to be notified when the results become publicly available.

2 Background

In this section, we provide an overview of the GPU architecture, programming models, and memory spaces. Note that while the concepts we describe are general to GPU computing platforms, we use NVIDIA's terminology for our descriptions.

2.1 GPU Basics

GPU architecture. Figure 1 shows an architecture overview of a typical GPU. The basic processing units in a GPU are called streaming multiprocessors (SMs). Each SM has a set of simple cores. With these cores, an SM can execute a group of parallel threads (known as a warp) in a Single-Instruction Multiple-Thread (SIMT) fashion. Modern GPUs usually have tens to hundreds of SMs. With a typical warp size of 32, a GPU can run thousands of threads simultaneously.

Each SM in a GPU contains its own register file, consisting of general-purpose registers and special registers. The general-purpose registers are partitioned among the threads that run on the SM. For example, in NVIDIA Ampere GPUs, every thread in an SM has its own 256 general-purpose registers, labeled from `R0` through `R255` [12, 46]. These registers temporarily store data that threads need immediate access to, such as variables or intermediate computation results. On the other hand, special registers have different roles and are used for specialized tasks. For example, `CLOCK` provides the current clock cycle count. Unlike general-purpose registers, some of the special registers are shared among all threads in the SM.

To serve the memory bandwidth demands of a large amount of threads, a GPU has its dedicated memory system, as shown in Figure 1. Each SM has its own private L1 cache and shared memory. SMs are connected to the shared L2 cache through a hierarchical on-chip network; The L2 cache is further connected to memory controllers which interface with the off-chip device memory. Similar to host memory on the CPU side, device memory is also based on DRAM. Currently, GDDR6 and HBM2 are the two most widely used DRAM types in

client and server GPUs, respectively. Note that the memory systems of the CPU and GPU are independent of each other. Before a program starts running on the GPU, the GPU driver loads the corresponding code into the device memory. Similarly, the data required by the GPU program must also be transferred to the device memory (before the data can be accessed). This is typically done through explicit operations in the program, although there are instances where the driver manages this data transfer implicitly [1].

```
1  /** device function **/
2  __device__ void add(char* d_global){
3      d_global[0] += 1;
4  }
5
6  /** CUDA kernel **/
7  __global__ void mem_type(char* d_global){
8      char d_local[10];
9      d_local[0] = d_global[0];
10     __shared__ d_shared[10];
11     d_shared[0] = d_global[0];
12     add(d_global);
13 }
14
15 /** CPU function, calling the cuda kernel **/
16 void kernel_launch (char* d_cpu){
17     char* d_global;
18     /** Allocate a GPU buffer **/
19     cudaMalloc(&d_global, 1024);
20     cudaMemCpy(d_cpu, d_global, 1024,
21                cudaMemcpyHostToDevice);
22     mem_type <<<8,32>>> (d_global);
23 }
```

Listing 1: An example CUDA program that uses multiple GPU memory spaces.

GPU virtual memory management. Modern GPU memory is virtualized, operating on a paging system. When SMs generate virtual addresses, the memory management unit (MMU) on the GPU performs virtual-to-physical address translation using the GPU page tables. Each running GPU program (i.e., a GPU context) has one page table. These page tables (from different active GPU contexts) are stored in the GPU memory and are regulated by the GPU driver [58][1]. Similar to CPU page tables, a GPU page table also has multiple levels: given a virtual memory address, the GPU MMU walks through these levels to find the page table entry (PTE) that contains the desired translation information. Prior work [58] has shown that recent NVIDIA GPUs use 5-level page tables. During a page table walk, a 49-bit virtual address is segmented, and its parts are used to select the walking path through the hierarchy.

2.2 GPU Programming and Execution

GPU programming model. GPUs were originally designed to accelerate graphics and multimedia processing; they could only be programmed using certain APIs such as OpenGL [53] and DirectX [36] to support 2D/3D graphics rendering. As the demand for utilizing GPUs in non-graphics computing tasks increases, various general-purpose GPU programming models have emerged. Of these, CUDA stands out as arguably the most successful and broadly used [41]. Listing 1 presents a simple CUDA program. Within the context of a CUDA program, there are several specific terms:

- **GPU kernel.** A kernel (line 7) is a function that is executed on the GPU and can be invoked from the host CPU. A CUDA program may consist of one or more kernels. To invoke a kernel, the GPU driver sends a corresponding kernel launch command to the GPU. It will first create a grid of thread blocks, with each block containing a certain number of threads. These thread blocks are then scheduled onto the available SMs on the GPU. When launching a kernel, the host code needs to specify the desired number of thread blocks and threads. For example, Listing 1 launches a kernel with 8 thread blocks and 32 threads in each block (line 22).
- **Device function.** A device function is a function that can only be called from kernels or other device functions. It can only be executed on the GPU (line 2).

NVIDIA PTX and SASS. PTX is an intermediate-level instruction set for NVIDIA GPUs that remains stable across different GPU generations. CUDA code is first compiled into PTX, and PTX is further compiled down to SASS, the low-level assembly language for NVIDIA GPUs. SASS instructions directly execute on NVIDIA GPU hardware. These instructions are tailored to the specific architecture of the GPU; different GPU generations may use different SASS instructions.

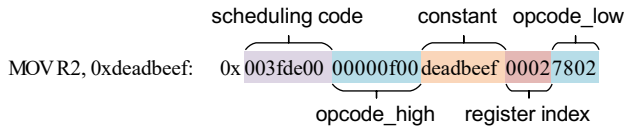

Figure 2: The encoding of the `MOV` instruction on Volta GPUs.

SASS instruction encoding. NVIDIA GPUs use a fixed-length instruction encoding format. Originally, the instruction length was 8 bytes. Starting with the Volta generation (released in 2017), the instruction length has been extended to 16 bytes. Unlike CPUs that typically use hardware-based instruction scheduling, NVIDIA GPUs delegate this scheduling task to the compiler. On Volta and later GPUs (with 16-byte instructions), the scheduling codes are embedded into the higher bits of each individual instruction, which specify the minimum wait time between consecutive instructions to meet dependency constraints. Figure 2 shows the encoding of the `MOV` instruction on Volta GPUs.

2.3 GPU Memory Spaces

Most GPU programming models allow memory allocation in different memory spaces, each of which has its unique behavior. Listing 1 shows a CUDA program that uses global,

[1]The GPU driver also maintains a copy of the page tables in host memory.

local, and shared memory, which are the most frequently used memory spaces in CUDA programs. The specific features of these memory spaces are shown in Table 1. We omit the discussion of other memory spaces, such as texture memory, as they are not related to our study.

- **Global memory** is managed by the GPU driver. Buffers in global memory can only be allocated by CPU code (before kernel launches) through driver API calls (line 19 in Listing 1). Global memory resides in the GPU's off-chip device memory; it can be cached in both the L1 and L2 caches. A buffer in global memory is accessible to all the threads in all the kernels of the program, until the buffer is freed. The load and store operations for global memory are usually performed using the instructions `LDG` and `STG`, respectively. In Listing 1, `d_global` is a buffer in global memory.
- **Local memory** is private to each thread. It is used to store the stack of a thread and is thus also called stack memory. Similar to global memory, local memory also resides in the device memory (and the caches). However, unlike global memory, data in local memory do not persist across kernels (since they are thread-private). The instructions `LDL` and `STL` are particularly used for local memory. `d_local` in Listing 1 is a buffer stored in local memory.
- **Shared memory** is a scratchpad memory region. It is shared among all the threads within the same thread block. As shown in Figure 1, shared memory is on-chip. Developers can place the data that is accessed frequently by threads in the same block into shared memory, in order to avoid the slow global memory access. Data in shared memory are not backed up in the off-chip device memory. `LDS` and `STS` are used for shared memory operations. `d_shared` in Listing 1 resides in shared memory.

In addition to the instructions mentioned above that are used for each particular memory space, there are also generic load and store instructions `LD` and `ST`, which can be used for accessing all the memory spaces.

3 GPU Memory Safety

3.1 Prior Art

Since CUDA is an extension of C/C++, CUDA programs can also have memory vulnerabilities similar to those in C/C++ programs. Several prior studies [19, 38, 48] have revealed that some of the memory errors found in CPU programs, such as buffer overflows, can also occur in CUDA programs. However, these studies have some limitations, which we explain below.

First, the investigations presented in these studies are fundamental and do not delve deeply into the problems particular to GPUs. For example, as explained in Section 2.3, CUDA features multiple, distinct memory spaces (unlike C and C++). Prior studies have only identified that buffer overflow errors can occur within a specific memory space. For instance, an OOB operation on a local memory buffer can compromise other data stored in that same local memory. However, they have not explored whether such an OOB operation can affect data in a different memory space.

Table 1: The specification of the pointers in Listing 1.

Pointer	Memory type	Storage	Cached	Load/store instructions	Scope
d_global	Global memory	Device memory (off chip)	Yes	`LDG/STG`	Process
d_local	Local memory	Device memory (off chip)	Yes	`LDL/STL`	Thread
d_shared	Shared memory	Shared memory (on chip)	No	`LDS/STS`	Thread block

Second, given that these studies were carried out several years ago, they only examined older GPUs (before Volta). However, NVIDIA has made significant changes to their GPUs since the Volta architecture [25]. For example, a new ISA has been introduced, where the instruction length was changed from 8 bytes to 16 bytes. Therefore, conclusions from these studies may not apply to newer GPU architectures. For example, these prior studies have two common conclusions. 1) Exploiting buffer overflow to hijack control flow in CUDA is very difficult because the return address is stored in an undisclosed memory location, not on the stack. 2) Traditional code injection attacks cannot be applied against CUDA programs because code and data are separated in memory. However, through our analysis (in Section 4), we found that **their conclusions do not hold.**

3.2 Our Goal

As GPUs have become a major computing component these days, it is important to understand the security problems that exist in modern GPUs. Our objective is to present an in-depth analysis of buffer overflow vulnerabilities on these computing devices, shedding light on the hidden threats that have been overlooked for years.

4 Demystifying GPU Memory

In this section, we provide a detailed analysis of GPU buffer overflows, addressing the limitations mentioned in Section 3. To gain a thorough understanding of the GPU memory model, we develop a tool using Direct Memory Access (DMA) to dump the content of device memory. We further manage to recover the page tables stored in device memory. NVIDIA has made their driver source code public. Therefore, from the driver code section concerning GPU page management [2] (and other NVIDIA documents [45]), we can obtain the overall format of the page table on NVIDIA GPUs. This allows us to identify and reconstruct the page table from the extracted device memory. Note that unless specified otherwise, all the experiments in this section are conducted on a system

with an NVIDIA GeForce RTX 3080 GPU, NVIDIA driver 470.63.01, CUDA 11.4, and the Ubuntu 20.04 OS. To simplify the analysis, we turn off CUDA ASLR.

4.1 Buffer Overflows across Memory Spaces

Here we study the buffer overflow issues in CUDA programs. As explained in Section 3, prior studies (e.g., [38]) have already shown that buffer overflows can cause memory corruption within a single memory space. Thus, we focus more on investigating the impact of buffer overflows across different memory spaces. Specifically, we first explore this problem between local and global memory, and then extend the study to cover the problem between these two and shared memory.

4.1.1 Accesses to Global and Local Memory

Global memory accesses are relatively straightforward. Global memory is indexed using a 49-bit virtual address (which is stored as a 64-bit value). There are two primary ways to access global memory: using the specialized `LDG/STG` instructions or the generic `LD/ST` instructions (cf. Section 2.3). These two pairs of instructions operate in a similar manner: as demonstrated in Listing 2, the target virtual memory address of the instruction is stored in a 64-bit register, which is formed by combining two 32-bit registers. We are not aware of any fundamental differences between accessing global memory using these two pairs of instructions. The choice of which pair to use appears to be based on the compiler's preference. Empirically, we observe that when the debugging option is enabled, the NVCC compiler always chooses `LD/ST`, otherwise it prefers `LDG/STG`.

```
/** R6: 0xcda00000 **/
/** R7: 0x7fff **/
/** R6.64 means R7||R6 **/
LDG R8, [R6.64]
STG [R6.64], R8

LD R8, [R6.64]
ST [R6.64], R8
```

Listing 2: Code for accessing global memory.

```
/** R6: 0xfffd80 **/
LDL R8, [R6]
STL [R6], R8

/** R6: 0xf2fffd80 **/
/** R7: 0x7fff **/
LD R8, [R6.64]
ST [R6.64], R8
```

Listing 3: Code for accessing local memory.

As local memory is private to individual threads, local memory accesses are more complicated than global memory accesses, as detailed below.

Instructions. Much like global memory, there are also two sets of instructions for accessing local memory, `LDL/STL` and `LD/ST`. However, they work very differently. As shown in Listing 3, `LDL/STL` uses a 24-bit address[2] stored in a 32-bit register. In contrast, as explained earlier, `LD/ST` requires a 49-bit virtual address (from a 64-bit register). In fact, we found that when `LD/ST` are used to access local memory, the 49-bit address appears to be the 24-bit address prefixed with a 25-bit value (which is `0x7ffff2` on our system). Note that virtual addresses belonging to other memory spaces *never* begin with this prefix value on our machine. Unlike global memory, the compiler always prefers to access local memory with `LDL/STL`, even when the debug flag is on.

```
1  __global__ void local_arr(){
2      uint32_t arr[10];
3      for(int i = 0; i < 10; i++)
4          arr[i] = 0xdead0000+threadIdx.x;
5      uint32_t* ptr = arr;
6      printf("thread %u addr %p data %u\n",
7              threadIdx.x, ptr, ptr[0]);
8  }
9  int main(){
10     cuda_kernel<<<1,32>>>();
11     return 0;
12 }
```

Listing 4: A simple CUDA program that allocates buffers in local memory.

Memory layout. We use the program in Listing 4 to explain the local memory layout: the CUDA kernel (`local_arr`) is launched with 32 threads per thread block and just one thread block overall. In this kernel, every thread allocates a local array (`arr`), resulting in 32 individual arrays in total. According to NVIDIA, each thread is only able to access its own array, but not the arrays that belong to other threads.

To understand how local memory is stored in the device memory, we execute the program in Listing 4 and pause it at line 6. Then, we dump the device memory content and identify the location of `arr` within it (by the data pattern). Figure 3 (b) shows a segment of the dumped device memory where `arr` resides. From this figure, we can observe that the local memory of different threads appears interleaved in the device memory. Specifically, the device memory sequentially stores `arr[0]` from all threads, then `arr[1]` from all threads, and so forth. We then conduct further experiments, adjusting the total thread count and the array's data type. These experiments reveal that every 32 bits of local memory from threads in a warp (comprising 32 threads) are always stored contiguously within the device memory. For variables larger than 32 bits, they are split into 32-bit segments and stored separately. Note that this layout information can help an attacker deliberately tamper with the local memory data, which we will show later.

Addressing. Given that each thread has its own private `arr`, one might naturally expect that each `arr` would have a distinct virtual address, which is similar to the scenario on CPUs. However, when we print the address of `arr[0]` (or `ptr[0]`) as done in line 6 of Listing 4, we have two interesting observations, as shown in Figure 4. First, `arr[0]` of different threads actually have the *same virtual address*. This virtual address is `0x7ffff2fffd80` on our machine (prefix+local memory address, cf. Listing 3). Second, when performing a data access using this address, each thread retrieves different data. More specifically, for a given thread, the retrieved data corresponds

[2] We found that the local memory for each thread is indexed with a 24-bit address using CUDA-GDB. This observation aligns with the findings from prior research [57].

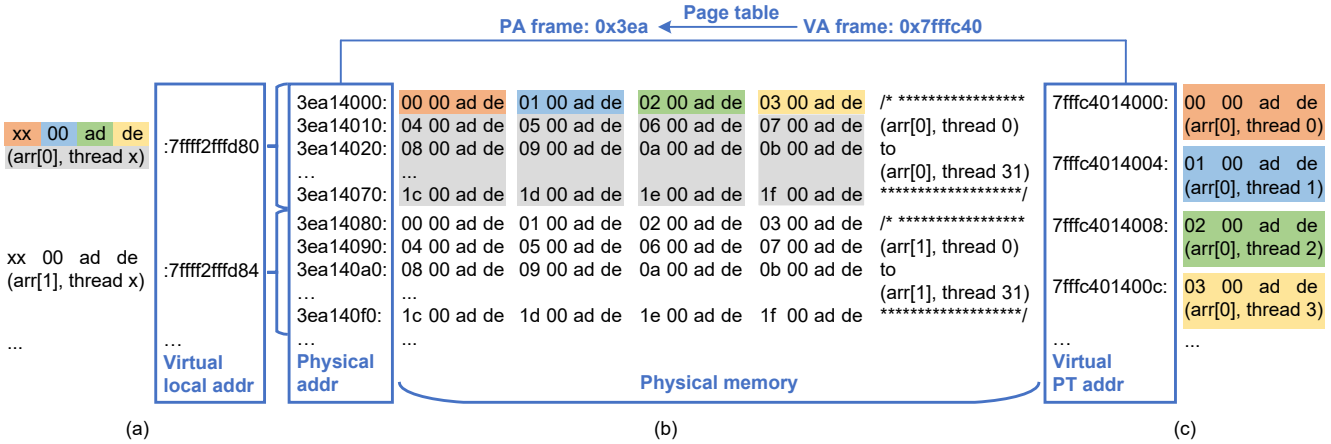

Figure 3: The mapping details of local memory: (a) the translation between virtual local addresses and physical addresses; (b) the layout of local memory (in device memory); (c) the translation between virtual PT addresses and physical addresses.

to `arr[0]` of that thread. With these results, we believe the mapping between the virtual and physical addresses of local memory aligns with the one depicted in Figure 3 (a) and (b): the physical address that a virtual address points to may vary, depending on the ID of the thread accessing this virtual address.

A natural question here is how the same virtual address is translated to different physical addresses (for different threads). While the specifics of local memory address translation on NVIDIA GPUs remain undisclosed, we give a conjecture here: there is likely a unique address translation mechanism for local memory, which is based on both the address and the thread ID. Moreover, the GPU hardware is able to recognize that a given memory operation is targeting the local memory (instead of other memory spaces) based on the given virtual address or the instruction. Specifically, it is considered a local memory operation if 1) the instruction is `LDL/STL` or 2) the instruction is `LD/ST` and the address begins with a certain pattern (e.g., `0x7ffff2`).

```
thread 0   addr 0x7ffff2fffd80   data 0xdead0000
thread 1   addr 0x7ffff2fffd80   data 0xdead0001
thread 2   addr 0x7ffff2fffd80   data 0xdead0002
...
```

Figure 4: The output of line 6 in Listing 4.

Upon examining the page table of the program in Listing 4, we have a very interesting observation. The aforementioned virtual addresses for local memory data (e.g., `0x7ffff2fffd80` for `arr[0]`) *are not mapped to* any valid physical addresses, according to the page table.[3] Instead, for each data block in the local memory, a disparate virtual address (e.g., `0x7fffc4014000` for thread 0's `arr[0]`)–seemingly unrelated to the unmapped virtual address above–is mapped to the physical address of this data block, as shown in Figure 3 (c). Importantly, when using this virtual address to access local memory, every thread gets the same data when accessing the same virtual address. For example, any thread accessing the address `0x7fffc4014000` will retrieve the value of thread 0's `arr[0]` (`0xdead0000`), regardless of its thread ID. Similarly, using the address `0x7fffc4014004`, every thread gets the value of thread 1's `arr[0]` (`0xdead0001`).

From the above results, we have two conclusions. First, for each physical address in local memory, there are two virtual addresses that can be used to access this physical address. Only one of them has a valid mapping in the page table; we refer to this virtual address as *the virtual PT address* (e.g., `0x7fffc4014000` in Figure 3 (c)), and refer to the other virtual address as *the virtual local address* (e.g., `0x7ffff2fffd80` in Figure 3 (a)). Second, as explained above, when accessing a virtual local address, the GPU hardware can recognize that this address is targeting the local memory and triggers a special address translation routine for it. This special routine takes the thread ID into account, and thus ensures that each thread can only access its own local memory. However, when accessing a virtual PT address, the GPU hardware does not recognize it as a local memory access, and therefore does not employ this special translation routine. Once knowing this address, one thread can access/modify the local memory of another thread in the program.

To further validate that both the virtual local address and the virtual PT address point to the same physical address, we conduct a read-after-write experiment. First, we write to `arr[0]` of thread 0 using the virtual PT address. Then, we read `arr[0]` of thread 0 using the virtual local address. We found that we can only read out the previously written value if we access another buffer whose size is at least 128B between our write and read operations. Given that the L1 Dcache on our GPU is 128B and L1 is indexed using virtual addresses and tagged using physical addresses, we believe that these

[3] This also confirms that there is a special address translation path for local memory addresses.

Table 2: Summary of the buffer overflow problem in CUDA; ✓ means the OOB operation can affect this memory space, while ✗ means it cannot.

OOB		Global mem				Local mem				Shared mem			
		Same thread	Same thread block	Same kernel	Same program	Same thread	Same thread block	Same kernel	Same program	Same thread	Same thread block	Same kernel	Same program
Global mem	LDG/STG	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗
	LD/ST	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✗	✗
Local mem	LDL/STL	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗
	LD/ST	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✗	✗
Shared mem	LDS/STS	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✗	✗
	LD/ST	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✗	✗

two virtual addresses are linked to the same physical address.

> Takeaway 1: On NVIDIA GPUs, each data block in local memory is linked to two virtual addresses; one of these addresses allows a CUDA thread to access/-modify the local memory of other threads.

4.1.2 Overflows across Global and Local Memory

OOB global memory references. Recall that global memory operations always use a 64-bit address, regardless of the instruction used. Therefore, attackers could exploit a buffer overflow vulnerability in global memory to influence any location in the device memory, including the local memory. Specifically, when accessing a global memory buffer with an OOB index, the attacker can manipulate the index to direct the target address (base address + index) towards either 1) the virtual PT address or 2) the virtual local address of the data in the local memory. Here we discuss the feasibility of these two approaches:

- **Accessing a virtual PT address**. As mentioned earlier, there are two sets of instructions for global memory references, `LD/ST` and `LDG/STG`. From our experiments, providing any of these instructions with a virtual PT address causes them to execute as expected, accessing the data in the local memory (of any thread) with the given address.
- **Accessing a virtual local address**. When providing a virtual local address to `LDG/STG`, it prompts a runtime error citing an illegal memory access. In contrast, when providing such an address to `LD/ST`, the instruction executes without any error. However, as explained before, using a virtual local address prevents us from modifying or accessing data belonging to other threads. We discuss this approach only for completeness. In real-world scenarios, the attacker would likely choose the former approach.

In short, a buffer overflow error in global memory may allow an attacker to target a virtual PT address, granting the attacker potential access to, or the ability to modify, the local memory data of any active thread in the program.

OOB local memory references. We found that there is a substantial gap between the virtual local addresses (of the local memory) and the virtual addresses of the global memory. On the tested system, the minimum difference between such addresses is `0x10000000`. With this discrepancy, whether an OOB operation on local memory can affect global memory actually depends on the specific instruction handling the operation: when using `LDL/STL`, which operates with a 24-bit address (the lower 24 bits of the virtual local address), OOB operations on local memory cannot affect global memory. In contrast, if `LD/ST` which takes the full 64-bit virtual local address is used, an OOB operation on local memory has the potential to fetch/modify data in global memory.

4.1.3 Overflows across Shared Memory and Local/-Global Memory

Data in shared memory is accessed in a similar manner to data in local memory. Specifically, shared memory can be accessed with either 1) the specialized instructions `LDS/STS`, using a 24-bit address, or 2) the generic instructions `LD/ST` using a 49-bit address. Again, the 49-bit address is formed by adding a prefix to the 24-bit address, which is `0x7ffff4` on the tested system. Consequently, when using the `LD/ST` instructions, an OOB operation on data in shared memory may affect the data in global or local memory. In addition, an OOB operation on data in global or local memory (with `LD/ST`) can affect the data in shared memory. Note that, unlike local memory, shared memory does not have a virtual PT address, since it is not part of the device memory. This means, accesses to shared memory remain confined to their legitimate scope (within the thread block). We cannot utilize virtual PT addresses to perform out-of-scope shared memory accesses (as done for local memory).

4.1.4 Summary

We provide a comprehensive summary of the overflow problem in CUDA in Table 2. First, with respect to the memory space, when using the generic memory instructions `LD/ST`, the problem can occur within a single memory space or across different spaces. In contrast, when using the specialized instructions (e.g., `LDL/STL`), the problem is restricted to a single memory space. An exception is that, OOB global memory references with `LDG/STG` can influence local memory.

Second, in terms of memory scope (i.e., visibility), when targeting local memory, an overflow error can result in ac-

cesses beyond the intended scope. This is due to the use of virtual PT addresses. In other scenarios, memory accesses are always confined to the legitimate scope.

Notice that the conclusions presented in this section, particularly those related to how GPU memory accesses are managed, are based on extensive reverse engineering efforts. While they are supported by thorough experimentation, we cannot claim with absolute certainty that our findings are entirely accurate. However, it is crucial to note that the primary objective of our reverse engineering analysis is not to perfectly reconstruct the GPU memory access functionality, but rather to investigate the potential for buffer overflow vulnerabilities in CUDA programs. Despite any potential inaccuracies in our reverse engineering results, we have conclusively demonstrated that buffer overflows on GPUs can be exploited to access/modify data across different memory spaces and beyond legitimate scopes (Table 2).

```
/** Start of the func **/
/** RZ is always 0 **/
IADD3 R1, R1, -0x70, RZ ;
STL [R1+0x68], R25 ;
STL [R1+0x64], R24 ;
STL [R1+0x60], R23 ;
STL [R1+0x5c], R22 ;
STL [R1+0x58], R21 ;
STL [R1+0x54], R20 ;
STL [R1+0x50], R19 ;
STL [R1+0x4c], R18 ;
STL [R1+0x48], R17 ;
STL [R1+0x44], R16 ;
STL [R1+0x40], R2 ;
```

```
/** End of the func **/
LDL R2, [R1+0x40] ;
LDL R16, [R1+0x44] ;
LDL R17, [R1+0x48] ;
LDL R18, [R1+0x4c] ;
LDL R19, [R1+0x50] ;
LDL R20, [R1+0x54] ;
LDL R21, [R1+0x58] ;
LDL R22, [R1+0x5c] ;
LDL R23, [R1+0x60] ;
LDL R24, [R1+0x64] ;
LDL R25, [R1+0x68] ;
IADD3 R1, R1, 0x70, RZ ;
RET.ABS.NODEC R20 0x0 ;
```

Figure 5: Assembly code when entering/leaving the device function; the 64B local array in the device function is stored in `[R1]` to `[R1+0x3c]`.

4.2 Return Address Corruption

Return address corruption is a severe security threat as it allows an attacker to hijack a program's control flow, potentially leading to arbitrary code execution. On CPUs, an attacker can exploit a stack buffer overflow vulnerability to overwrite the return address on the stack. However, prior studies [38, 48] suggest that such exploitation is not feasible on GPUs: they claim that on GPUs the return address is stored in an undisclosed location in the device memory, rather than on the stack (local memory). We reexamine this claim in this section.

4.2.1 Stack Management

To understand the management of the return address on GPUs, we launch a simple CUDA kernel whose only task is to call a device function. The device function allocates a 64B local array and fills it with `0xdeadbeef`. Figure 5 shows assembly code snippets of this device function, illustrating the stack management at the beginning and end of the function. When the function starts, certain registers that the function intends to modify are pushed to the stack to be later restored upon the function's completion. Conversely, after the function finishes, these registers are popped from the stack and restored. These code snippets provide two key insights into CUDA's stack management:

- **The role of `R1`.** In CUDA, `R1` is a general-purpose register, not a special register [44]. However, the above code implies that `R1` can be used as the stack pointer. In fact, by further examining common CUDA libraries such as `libcudnn`, we found that `R1` is the only register that has been used as the stack pointer. Notably, NVIDIA's list of special registers does not contain any register for stack pointer [44].
- **Stack commands.** Similar to the RISC-V architecture, NVIDIA GPUs do not have dedicated `push/pop` instructions. Instead, a push operation is achieved through a local memory write together with a decrement of the stack pointer. Conversely, a pop operation is achieved by a local memory read and an increment of the stack pointer.

Return address. In Figure 5, the return instruction (`RET`) uses `R20` as an operand. This register is pushed to the stack upon entering the function, and retrieved right before the `RET` instruction. Intuitively, the value in `R20` should be related to the return address. To validate this, we run the program in CUDA-GDB and we found that the value of `R20.64` (i.e., `R21||R20`)[4] is the same as the expected return address (which in our specific case is `0x7fffd6fad8e0`). To better understand this, we extract the device memory (before `RET` is executed) and show the local memory section in Figure 6. We can see that the value of `R20.64` is located in the local memory, near the local array (filled with `0xdeadbeef`). This observation confirms that the return address, represented by `R20.64`, is stored together with the local variables on the stack, in contradiction to the conclusions of previous studies.

```
4d612a00: ef be ad de  ef be ad de  ef be ad de  ef be ad de
*
4d613200: ff 7f 00 00  ff 7f 00 00  ff 7f 00 00  ff 7f 00 00
*
4d613280: 00 00 a0 d7  00 00 a0 d7  00 00 a0 d7  00 00 a0 d7
*
4d613300: 01 00 00 00  01 00 00 00  01 00 00 00  01 00 00 00
*
4d613380: 10 00 00 00  10 00 00 00  10 00 00 00  10 00 00 00
*
4d613400  01 00 00 00  01 00 00 00  01 00 00 00  01 00 00 00
*
4d613480: e0 d8 fa d6  e0 d8 fa d6  e0 d8 fa d6  e0 d8 fa d6
*
4d613500: ff 7f 00 00  ff 7f 00 00  ff 7f 00 00  ff 7f 00 00
```

Figure 6: Part of the local memory in the dumped device memory; the local array and `R20.64` (i.e., `R21||R20`) are highlighted; "*" means the data is the same with above.

[4] While not explicitly specified in the instruction, `RET` appears to always retrieve the return address from `R20.64` rather than just `R20`.

We further perform an OOB operation on the local array to overwrite the return address, pointing it into another function in the program. As anticipated, when the function returns, it proceeds with the overwritten address, executing the code located there, rather than returning to the original caller address. Similar behavior occurs when we perform an OOB operation on global memory (or shared memory) to overwrite the return address (cf. Table 2).

> Takeaway 2: In CUDA, exploiting a buffer overflow vulnerability allows an attacker to modify the return address stored in the local memory (stack), and therefore redirect the control flow of the program.

4.2.2 Return Address on the Stack

In Section 4.2.1, we focus on the scenarios where the return address register (e.g., `R20` in Figure 5) is pushed to the stack during the execution of the function. However, the NVCC compiler does not always opt for this approach. Rather than pushing the return address register to the stack, the compiler often avoids using that register throughout the function. It only chooses to push this register if it is challenging (or infeasible) to ensure that this register remains untouched in the function. Below are some scenarios where the return address register is pushed to the stack.

1. The device function is recursive.
2. The device function has a substantial number of local variables, resulting in insufficient registers (i.e., register spill-out).

4.3 Code Injection

Executing data pages. With the capability to overwrite the return address, the attacker can redirect the execution to a data page which they have filled with shellcode. Then, when the function returns, the execution is diverted to this malicious code, resulting in a code injection attack. Such attacks have already been mitigated on CPUs: for example, most CPU systems have the `W^X` policy implemented, which mandates that every memory page can be either writable or executable, but not simultaneously both. This prevents the shellcode from being directly executed. However, we found that this policy is not implemented on modern GPUs.

Our results show that by manipulating the return address, we can redirect the control flow to a global memory address and execute the data there (as code). We can also change the control flow to point to a local memory address and execute the data on the stack.[5] These findings suggest that GPUs do not make the writable data pages non-executable. We also found that according to the page table format [2, 45] released by NVIDIA, there is not an "executable bit" (nor a "dirty bit") in the PTE. This implies that GPUs do not check whether an address is a legitimate code address before executing its content. Note that prior studies claim that executing a data buffer is infeasible on GPUs; they believe that this is either because the code and data addresses are separated, or because the data pages are not executable [38, 48]. We found that neither of these hypotheses is accurate.

Modifying code pages. Given that data pages are executable, a natural question that arises is whether code pages are modifiable. In fact, prior work has already shown that it is possible to modify code pages on older GPUs. We further verify this on modern GPUs by examining the device memory before and after writing to a code page. Additionally, when inspecting the GPU page table format, we notice a "read-only bit" in the PTE. However, after analyzing the page table (extracted from the device memory), we found that this bit consistently remains unset, even for code pages.

> Takeaway 3: NVIDIA GPUs do not differentiate between code and data pages.

4.4 CUDA ROP

Return address corruption can also lead to code reuse attacks, of which ROP is a primary example. ROP has proven to be highly effective on CPUs, with numerous ROP gadgets found in commonly used library code, such as `libc`. Here we study the feasibility of ROP on modern GPUs.

CUDA library code. Upon inspecting the content of the device memory during the execution of a CUDA program, we found that, besides the application-specific code, there is additional code loaded into the device memory. This additional code is the same for every CUDA program. We compare this code with the machine code of common CUDA libraries and found that this code is part of `libcuda`. NVIDIA describes `libcuda` as the CUDA driver API library, which handles tasks related to direct interaction with the GPU, such as memory management, error handling, and stream management. Functions within this library include (but are not limited to) `printf`, `cuMemcpy`, and `cuMemFree`. In addition, our experiments show that after redirecting the control flow of a CUDA program to an address within this driver API code, we can execute this code without triggering any errors.

CUDA ROP gadgets. We examine this driver API code and found 190 return instructions (`RET`). Out of these `RET` instructions, only 52 are accompanied by an instruction that pops the return address. This means, there are only 52 possible ROP gadgets in this drive API code, consisting of 7 memory corruption gadgets and 45 others. Listing 7 shows two example gadgets: the first one writes data from a register to a mem-

[5] The control flow cannot be redirected to a shared memory address, meaning we cannot execute the data stored in shared memory as if it were code.

```
ST.E.64 [R28.64], R4 ; /** Store gadget **/
BSYNC B7 ;
LDL R0, [R1] ;
BMOV.32 B6, R27 ;
LDL R20, [R1+0x18] ;
LDL R21, [R1+0x1c] ;
LDL R2, [R1+0x4] ;
LDL R16, [R1+0x8] ;
LDL R17, [R1+0xc] ;
LDL R18, [R1+0x10] ;
LDL R19, [R1+0x14] ;
LDL R22, [R1+0x20] ;
LDL R23, [R1+0x24] ;
LDL R24, [R1+0x28] ;
LDL R25, [R1+0x2c] ;
LDL R26, [R1+0x30] ;
LDL R27, [R1+0x34] ;
LDL R28, [R1+0x38] ;
LDL R29, [R1+0x3c] ;
IADD3 R1, R1, 0x40, RZ ;
BMOV.32 B7, R0 ;
RET.ABS.NODEC R20 0x0 ;
```

```
LD.E.STRONG.GPU R5, [R6.64+0x5c] ; /** Load-and-store gadget **/
LOP3.LUT R0, R0, 0xffff, RZ, 0xc0, !PT; /** Logic operation **/
IADD3 R0, R0, 0x4, RZ ;
IMAD R4, R5, 0x20000, R0 ;
IMAD.MOV.U32 R5, RZ, RZ, RZ ;
ST.E.64 [R28.64], R4 ; /** Store **/
BSYNC B7 ;
LDL R0, [R1] ;
BMOV.32 B6, R27 ;
IMAD.MOV.U32 R4, RZ, RZ, R2 ;
LDL R20, [R1+0x18] ;
LDL R21, [R1+0x1c] ;
LDL R2, [R1+0x4] ;
LDL R16, [R1+0x8] ;
LDL R17, [R1+0xc] ;
LDL R18, [R1+0x10] ;
LDL R19, [R1+0x14] ;
LDL R22, [R1+0x20] ;
... /** Pop R23 to R29 **/
IADD3 R1, R1, 0x40, RZ ;
BMOV.32 B7, R0 ;
RET.ABS.NODEC R20 0x0 ;
```

Figure 7: CUDA ROP gadgets.

ory address, while the second one reads data from a memory address and then writes it to another address. Unfortunately, our analysis suggests that this CUDA gadget set is not Turing complete. However, later in Section 5 we show that even with these limited gadgets, the attacker can significantly reduce the performance of DNN-based applications on GPUs. In addition, using the memory corruption gadgets, we might be able to modify the CUDA code (which is not write-protected) to create a Turing-complete collection of gadgets, thus achieving arbitrary computation.

Note that it is difficult to have unintended ROP gadgets, since all the GPU instructions must be 8B aligned. In addition, including other common CUDA libraries, such as `libcublas` and `libcudnn`, does not really bring more ROP gadgets: these libraries are so optimized that the return address is almost never pushed to the stack; it is typically stored in a register instead.

> Takeaway 4: ROP can be used to read or write the memory on NVIDIA GPUs.

Generality. In this section, we discuss and present the investigations using the platform specified at the beginning of this section. However, the conclusions drawn from these investigations, including the feasibility of memory corruption across memory spaces (cf. Table 2), and the potential for code injection and code reuse attacks, are applicable to other modern NVIDIA GPUs as well.[6] We have verified these vulnerabilities on multiple NVIDIA GPUs spanning several recent architectures, including Volta, Turing, Ampere, and Ada Lovelace; we conduct experiments on these GPUs with various NVIDIA drivers ranging from version 470.63 (released in July 2021) to version 550.67 (released in March 2024), and multiple CUDA toolkits ranging from version 11.2 to version 12.4. The results consistently demonstrate that these vulnerabilities exist across all tested GPUs, regardless of the driver/CUDA version used. The complete list of the tested GPUs is provided in Appendix A.

[6]The specific details, such as the number of ROP gadgets, may vary slightly depending on the CUDA version.

5 Case Study: Corruption Attacks on DNN

In this section, we demonstrate how the GPU memory corruption vulnerabilities discussed in Section 4 can pose significant security risks for DNN-based applications, which are one of the most common GPU applications.

5.1 Threat Model

Victim. The victim is a DNN-based application running on a server equipped with a modern NVIDIA GPU. This application receives requests from remote users, processes these requests using a DNN model, and sends the responses back. We assume that some of the CUDA kernels involved in the process of DNN inference have memory corruption vulnerabilities, (the vulnerability examples will be discussed later in Section 5.2). As a common practice, model parameters, such as weights, are loaded into the device memory during application initialization. To minimize response latency, these parameters persist in memory across user requests, rather than being reloaded for each new request or removed after processing a request.

Attacker. The attacker is a remote user who can send requests to the victim application. By crafting a malicious request (detailed in Section 5.2), the attacker is able to exploit a buffer overflow vulnerability in the GPU kernels used by the victim application. The primary goal of the attacker is to alter the model parameters, such as the weights, through this vulnerability. Consequently, the inferences for future requests from other users will be compromised. We assume that the attacker has knowledge of the layout of the victim's DNN

```
/** Device func for maxtrix-vector multiplication **/
/** T: the input matrix **/
/** V: the input vector **/
/** R: the result vector **/
/** m: the number of rows in T **/
/** n: the number of columns in T **/
__device__ void matvecmul(scalar_t* T, scalar_t* V,
     scalar_t* R, int m, int n)
{
    scalar_t arr_local [64];
    int ncols = n/(blockDim.y*blockNum);
    int col0 = blockIdx.y*blockDim.y+threadIdx.y;
    for (int k=0; k<ncols; k+=1)
        arr_local[k] = R[col0*ncols+k];
    ...
}
```

Listing 5: The device function for matrix-vector mulpiclication with a buffer overflow vulnerability.

model weights.

5.2 Application Setups

Under the aforementioned threat model, we choose four widely used vision models as potential victims for our evaluation: ResNet-18 [26], ResNet-50 [26], VGGNet [54], and Vision Transformer (ViT) [35].[7] We implement these models with popular DNN frameworks in cloud environments, such as PyTorch [49].

We host the DNN inference application (i.e., the victim application) in a virtual machine on a cloud system equipped with a server-grade GPU, which is different from the system used for the experiments in Section 4. We utilize NVIDIA's virtual GPU (vGPU) technology to virtualize the GPU [13]. This is a common configuration for DNN inference in cloud environments [7, 8]. The detailed specifications of this system can be found in Appendix D. Note that **vGPU does not support CUDA ASLR**; with vGPU, addresses are not randomized even when ASLR is activated. We keep CUDA ASLR activated in our configuration as it is the default setting. However, it has no effect. We discuss CUDA ASLR in non-virtualized environments later in Section 6.2.

Previous work [18] has suggested that modern DNN frameworks may be vulnerable to GPU buffer overflow issues, but has not disclosed any specific instances. Similarly, we have not identified any overflow vulnerabilities in these frameworks. However, the goal of this paper is not to uncover such vulnerabilities; we leave that task for future research. Instead, for the purpose of our evaluation, we intentionally introduce buffer overflow vulnerabilities into the DNN frameworks.

Buffer overflow vulnerability. We include a CUDA device function for matrix-vector multiplication (with an overflow vulnerability) in the victim application, as shown in Listing 5. Matrix-vector multiplication is important and commonly used in DNN inference. This code uses multiple CUDA threads for each row in the matrix to calculate the partial sums, and each thread transfers the necessary portion of the vector from global memory to its local memory prior to the calculation. We deliberately introduce a vulnerability in this kernel: it lacks proper checks to ensure that the size of the vector portion handled by each thread does not exceed the capacity of the thread's local array. Consequently, a stack overflow can occur when the vector size, which is controlled by the user (explained below), is larger than expected.

Triggering the buffer overflow vulnerability. In order to trigger the vulnerability in Listing 5, we assume that the size of the vector (`n`) is controlled by the users. An example of this situation occurs during the data preprocessing stage. Specifically, the size of the input data provided by the user may not always match the required input size of the DNN model. For example, a user might provide an image of size 512×1024 pixels, while the DNN model is designed to process images of only 256×512 pixels. To handle such discrepancies, a convolution layer might be used to preprocess the user input before feeding it into the DNN model. This convolution layer often employs matrix-vector multiplication for performance optimization [16]. In this preprocessing convolution layer, the dimensions of the matrix and vector (`m` and `n`) are determined by the size of the user input. This potentially allows a user to trigger the vulnerability in Listing 5, if the input size is significantly larger than the size expected by the DNN model.

5.3 Attack Methods and Results

In this section, we examine the two strategies that an attacker can employ to modify the weights: code injection and ROP attacks. We discuss the specific steps an attacker must take to launch these attacks, as well as the resulting outcomes.

5.3.1 Code Injection Attack

We implement the code injection attack as a controlled weight attack: the attacker has control over the data written to memory and can modify the weights to any desired value. Specifically, the attacker prepares a data buffer with shellcode that writes specific values to given addresses, and uses a stack overflow vulnerability to redirect the control flow to this buffer (cf. Listing 5). Details of the shellcode can be found in Appendix C. There are three steps in the attack:

Step 1: The attacker prepares a data buffer whose size is large enough to trigger the buffer overflow error in the victim (cf. Listing 5) when this buffer is sent to the victim for DNN inference.

Step 2: The attacker manipulates the data in the buffer to achieve two critical objectives: 1) the local array of each thread (`arr_local` in Listing 5) is filled with expected shellcode after this buffer is copied to the local memory; 2) the return address of each thread is overwritten with the address of this local array (where the shellcode resides), after the data copy.

[7] We also test the memory corruption attacks on several large language models (LLMs) which we obtain from Hugging Face [10], the details can be found in Appendix E.

Table 3: The DNN inference accuracy with the weight modification attacks (only for weights in the first layer).

DNN model	Clean acc.	Weight modification (controlled)				Weight modification (uncontrolled)			
		10%	20%	50%	100%	10%	20%	50%	100%
ResNet-18 (CIFAR-10)	87.37%	10.00%	10.00%	10.00%	10.00%	87.14%	87.32%	81.65%	10.83%
VGG-19 (CIFAR-10)	83.56%	13.46%	13.46%	11.19%	9.66%	74.90%	62.77%	59.01%	10.00%
ResNet-50 (ImageNet-1K)	84.97%	58.21%	55.21%	54.75%	44.35%	84.93%	84.27%	80.64%	65.33%
ViT (ImageNet-1K)	93.64%	0.09%	0.08%	0.12%	0.09%	92.78%	90.56%	90.33%	88.54%

These preparations are crucial to ensure that when the buffer overflow is triggered, it leads to the expected code injection attack.

Step 3: The attacker initiates a DNN inference request using this data buffer as the input.

5.3.2 ROP Attack

We implement the ROP attack as an uncontrolled weight modification attack: the attacker repeatedly executes a single memory write gadget to modify the weights. The attacker controls the register providing the address used in the write operation, but does not control the register that supplies the data to be written. Details of the gadget are in Appendix B. Similar to the code injection attack introduced in Section 5.3.1, this ROP attack also has three steps, which are very similar to those in the code injection attack. However, the objectives in the second step, preparing the data buffer, are slightly different. Specifically, the attacker needs to manipulate the data in the input buffer to ensure that, after the data copy, 1) the return address on the stack is overwritten with the address of the ROP gadget, and 2) certain registers, which will be used by the ROP gadget, receive the expected values when popped from the stack.

5.3.3 Address of the Malicious Code

In the aforementioned two types of attacks, in order to modify the DNN weights, the attacker needs to know the address of the malicious code (either the ROP gadgets or the shellcode). As explained in Section 5.2, CUDA ASLR has no effect when using vGPU, making this address predictable and stable across executions. As a result, it is rather straightforward for the attacker to determine the address of the shellcode/ROP gadgets. For example, once the attacker profiles the memory of one NVIDIA GPU and identifies the addresses of the ROP gadgets, it can launch ROP attacks on all NVIDIA GPUs of the same generation and using the same CUDA toolkit, since these gadgets are loaded at fixed addresses. We discuss the scenarios in which ASLR takes effect (in native environments without virtual machines) in Section 6.2.

5.3.4 Results

We test the accuracy of the model after modifying the weights in the first layer of each model, using the attack methods in Section 5.3.1 and Section 5.3.2. Table 3 shows the accuracy results after modifying 10%, 20%, 50%, and 100% of the weights. In the code injection attack where the attacker can specify the desired weight value, we choose a large value for each weight. This is because most weight values in DNN models are very small; using a large value is expected to substantially impact the model performance. As shown in the table, this approach significantly reduces the accuracy across all tested models. ResNet-18 and ViT are especially affected, with the accuracy nearly mirroring random guesses. In contrast, in the ROP attack where the attacker does not have control over the modified weight value, the accuracy remains largely unaffected. This is because the ROP gadget used in the attack happens to change the weights to a small value, rather than a large value, which is close to the original values of the weights.

6 Discussion

6.1 `BSYNC` in CUDA ROP Gadgets

Usage of `BSYNC`. CUDA provides synchronization mechanisms at different levels. The `BSYNC` instruction is specifically used for intra-warp synchronization. Although threads in a warp are intended to execute the same instruction simultaneously, some scenarios, such as conditional branches, can lead to thread divergence. `BSYNC` and `BSSY` are used to manage such situations: `BSSY` signals the hardware to prepare for divergence and specifies the address for re-convergence [25]. `BSYNC` serves as the synchronization barrier: when a thread in the warp reaches `BSYNC`, it waits for other threads in the warp.

We found that `BSYNC` and `BSSY` always appear together (in a pair) in CUDA binaries. However, this pairing might not be maintained in ROP gadgets. For example, the gadgets in Figure 7 contain only `BSYNC` but not `BSSY`. This means, the re-convergence address is not specified when `BSYNC` executes, which may cause an error. Interestingly, our analysis reveals that if the threads in a warp do not diverge, `BSYNC` does not influence the execution. Thus, as long as threads remain in sync when executing a ROP gadget, the gadget will function as expected without being affected by `BSYNC`. This is a feasible condition, especially for DNN-based applications, where thread divergence rarely occurs.

6.2 CUDA ASLR

In non-virtualized environments, CUDA supports ASLR for both data and code. In addition, the activation of CUDA ASLR

depends on the ASLR settings in the OS. If ASLR is enabled in the OS, CUDA ASLR is also enabled (by the GPU driver), reflecting a similar level of randomization (e.g., no, conservative, or full randomization [11]) as the one in the OS. Note that, as mentioned in Section 5.2, CUDA ASLR does not function in virtualized environments with the vGPU technology [13].

CUDA ASLR, when functioning, makes it more difficult for the attacker to obtain the address of the shellcode and thus can help mitigate code injection attacks. However, the attacker may be able to bypass CUDA ASLR through GPU side channels (e.g., [58]). In addition, we found that the CUDA driver API library mentioned in Section 4.4 is always loaded at a fixed address, even when full randomization is applied. Therefore, we cannot rely on CUDA ASLR to completely stop the attacker from exploiting the ROP gadgets in this library.

6.3 CUDA JOP

Similar to ROP, jump-oriented programming (JOP) [15] is also an advanced code reuse attack technique. Instead of chaining gadgets that end in a return, JOP makes use of gadgets that end in an indirect jump. These jumps use registers to determine the destination address. To chain the JOP gadgets we need a "dispatcher" (also called "dispatcher gadget"). Its role is to load the address of the next gadget into the appropriate register and then jump to it.

On NVIDIA GPUs, the indirect jump instruction is `BRX` (e.g., "`BRX R3, 0xa0`"). With this instruction, it is possible to form JOP on GPUs. As mentioned in Section 4.4, we did not find any ROP gadgets in common CUDA libraries such as `libcudnn` and `libcublas`. However, we found that some of these libraries do use `BRX` and therefore contain JOP gadgets. Unfortunately, after analyzing these gadgets, we found that they can only perform limited functionality. For example, the `libcuda.so.470.63.01` library contains 156 JOP gadgets, but 151 of these are merely for shifting register values. Similar observations have been made with other common CUDA libraries, such as `libcudnn_cnn_infer.so.8.2.2`. However, we may combine the ROP and JOP gadgets to achieve more functionality. More details of this can be found in Appendix F.

6.4 Memory Errors in Real-World GPU Applications

Memory errors, especially buffer overflow errors, have been identified in existing GPU applications. First, a previous study [22] analyzed 175 commonly used GPU programs in 16 benchmark suites and found 13 buffer overflow errors in 7 of these programs; some of these errors are very similar to the one we assumed in the ML attacks. Second, buffer overflow errors have also been found in web browsers that utilize GPUs to accelerate rendering tasks. For example, in 2022, researchers discovered a buffer overflow error in Chrome that occurs during the data transfer between CPU and GPU memory [3]. This allows a remote attacker to escape the sandbox through a crafted HTML page. In addition, in 2023, it was demonstrated that buffer overflows can also occur within WebGL programs running on GPUs, causing the browser to crash [4].

Furthermore, previous studies on fuzzing ML frameworks (e.g., [18]) have identified significant bugs in their GPU kernels. These include computation bugs, which lead to inaccurate results, and crash bugs, which can cause the entire application to crash. However, these preliminary fuzzing studies have not yet revealed any exploitable memory errors in these frameworks, which we leave for future work. Note that given the similar programming model between CUDA and C/C++, we believe it is likely that exploitable memory bugs exist in these frameworks.

6.5 CUDA Heap Exploitation

CUDA supports dynamic memory allocation: buffers dynamically allocated in CUDA kernels (using the `malloc()` function) reside in the GPU's heap memory.[8] The heap memory is persistent during the lifetime of a GPU process and is shared among the kernels in this process. NVIDIA has not released the specific memory allocator used in CUDA. However, our experimental results suggest that this allocator follows policies similar to those in CPU memory allocators (e.g., ptmalloc [14]). Consequently, CUDA programs may also be vulnerable to spatial/temporal heap exploits, similar to those found in CPU programs. However, the use of dynamic memory allocation on GPUs is generally discouraged due to its significant performance overhead [33]. After analyzing common CUDA applications and benchmarks, we did not find any scenario where dynamic buffer allocation is used. Thus, we believe that this issue is much less significant compared to overflows in memory that is statically allocated (local/global/shared memory).

6.6 Countermeasures

OOB detection tools. There have been many tools to statically/dynamically detect buffer overflow errors in GPU programs. First, NVIDIA Compute Sanitizer [40], a tool for GPU memory safety checks, is based on dynamic binary instrumentation: it intercepts every program instruction at runtime before the execution. While effective, this approach introduces considerable performance overhead. Second, cuCatch [57] is a compile-time tool that can help detecting both spatial and temporal memory errors during the execution of a CUDA program. It stores the necessary metadata for memory safety

[8] Heap memory is different from global memory. Global memory buffers can only be allocated by the CPU code (prior to a kernel launch, cf. Section 2.3), while heap buffers can be allocated by the GPU code during the execution of a CUDA kernel.

checks in a table, with one entry for each allocation. Given a pointer, the correpsonding table entry index can be either embedded in the upper bits of the pointer (if possible), or stored in the shadow memory. cuCatch introduces much less performance overhead compared to NVIDIA Compute Sanitizer. However, it is important to note that neither of these tools can achieve a 100% detection rate.

Stack cookies. Stack canaries or stack cookies are secret values that are placed between a buffer and control data on the stack to monitor for stack overflows. Unfortunately, NVIDIA has not adopted this technique on their GPUs. In addition, it is important to note that previous studies have already shown that the attacker might be able to determine the value of the stack cookies, thus bypassing the detection mechanism (e.g., [51]).

In fact, several canary-based detection tools (for GPU buffer overflows) have been proposed by researchers, such as GMOD [21], GMODx [20], and clARMOR [22]. They insert a canary before and after every GPU buffer (not just stack buffers) and detect buffer overflows by periodically verifying the integrity of these canaries. These tools have minimal performance overhead. However, they cannot detect OOB read operations or non-adjacent OOB read/write operations.

ASLR and PIE. Both Position Independent Executable (PIE) and ASLR are supported on NVIDIA GPUs. However, we found that the CUDA driver API library that contains powerful ROP gadgets is not compiled as PIE and is loaded at a fixed address even with CUDA ASLR activated (cf. Section 6.2). As a result, ASLR cannot thwart ROP attacks on GPUs. It is not clear why NVIDIA has chosen this design approach. To effectively prevent ROP attacks, it is crucial for NVIDIA to compile this library as PIE and ensure that it is subject to ASLR. In addition, ASLR makes it more difficult for the attacker to launch code injection attacks. However, the attacker might be able to bypass ASLR through side-channel attacks (cf. Section 6.2).

W^X policy. Differentiating between code and data pages is critical for counteracting code injection attacks. As mentioned in Section 4, there is already a read-only bit in the PTE. To understand whether this bit takes effects, we modified the bit through IOMMU during the execution of a CUDA program, and found that setting this bit effectively prohibits any modifications to the page. Thus, the GPU driver can set this bit for code pages to prevent code modification. In addition, an executable bit needs to be included in order to prevent the execution of data pages.

6.7 Related Work

Research in memory vulnerabilities on GPUs has been very limited. First, in 2016, Miele conducted a preliminary exploration into buffer overflow vulnerabilities in CUDA [38]. This research shows that, on a GTX TITAN Black GPU (Kepler architecture with sm_30), it is possible to exploit a stack overflow to overwrite function pointers, therefore redirecting the execution flow. It further shows that the function call and return mechanism on these GPUs is based on a `PRET` instruction followed eventually by a `RET` instruction. The `PRET` instruction stores the return address in an *unknown location*, making the traditional ROP impractical on these GPUs. In addition, this research concludes that code and data address spaces are separated and thus executing data buffers is not possible. Note that this work also confirmed the feasibility of CUDA heap overflows, although heaps are rarely used in CUDA programs [33].

Second, in the same year, Di et al. conducted similar experiments on a GeForce GTX 750Ti GPU (Maxwell architecture with sm_50) [19], reaching conclusions that align with those from Miele. However, compared to Miele's study, this work provides a much more detailed analysis on GPU heap overflows.

Third, later in 2021, Park et al. performed a deeper analysis of GPU memory exploitation techniques. They presented the first attack on DNN frameworks based on GPU memory manipulation [48]. The experiments in this work were carried out on a GTX 950 GPU (Maxwell architecture with sm_50) and a GTX 1050 GPU (Pascal architecture with sm_60). This work reached several conclusions that are the same as prior work, such as the hidden return address. However, it also made a new discovery that the code pages are writable on NVIDIA GPUs.

7 Conclusion

In this paper, we present a comprehensive study of the buffer overflow issues in CUDA programs. First, we reverse engineered the mechanisms used to access various GPU memory spaces, demonstrating that buffer overflow errors can cause memory corruption across memory spaces and exceed data's legitimate scopes. Second, we explored the code and data management policies on GPUs, revealing that traditional code injection attacks remain functional on GPUs. We also analyzed the mechanics of function returns and proved the feasibility of CUDA ROP. Finally, we demonstrated that the vulnerabilities discovered in this paper pose significant security risks to DNN applications running on GPUs. The Proof-of-Concept for CUDA code injection and CUDA ROP is available at https://github.com/SecureArch/gpu_mem_attack.

Acknowledgement

The authors thank the anonymous USENIX Security 2024 reviewers and shepherd for their valuable comments and suggestions that help us improve the quality of the paper. This work is supported in part by the US National Science Foundation (CCF-2334628, CCF-2154973, CCF-2011146, and CNS-2147217).

References

[1] https://developer.nvidia.com/blog/unified-memory-cuda-beginners/.

[2] https://github.com/NVIDIA/open-gpu-kernel-modules/blob/main/kernel-open/nvidia-uvm/uvm_pascal_mmu.c.

[3] CVE-2022-4135. Available at https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-4135.

[4] CVE-2023-4582. Available at https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-4582.

[5] Flan-T5. Available at https://huggingface.co/docs/transformers/en/model_doc/flan-t5.

[6] Four data cleaning techniques to improve large language model (LLM) performance. Available at https://medium.com/intel-tech/four-data-cleaning-techniques-to-improve-large-language-model-llm-performance-77bee9003625.

[7] GPU optimized virtual machine sizes. Available at https://learn.microsoft.com/en-us/azure/virtual-machines/sizes-gpu.

[8] GPU platforms. Available at https://cloud.google.com/compute/docs/gpus.

[9] High-throughput generative inference of large language models with a single GPU.

[10] Hugging Face. Available at https://huggingface.co.

[11] Linux and ASLR. Available at https://linux-audit.com/linux-aslr-and-kernelrandomize_va_space-setting/.

[12] NVIDIA GPU Microarchitecture. Available at https://www.ece.lsu.edu/gp/notes/set-nv-org.pdf.

[13] NVIDIA virtual GPU software documentation v17.0 through 17.1. Available at https://docs.nvidia.com/grid/17.0/index.html.

[14] The GNU C Library. Available at https://www.gnu.org/software/libc/manual/html_mono/libc.html.

[15] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM symposium on information, computer and communications security (ASIACCS)*, 2011.

[16] Kumar Chellapilla, Sidd Puri, and Patrice Simard. High performance convolutional neural networks for document processing. In *Tenth international workshop on frontiers in handwriting recognition*, 2006.

[17] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[18] Neophytos Christou, Di Jin, Vaggelis Atlidakis, Baishakhi Ray, and Vasileios P Kemerlis. IvySyn: Automated vulnerability discovery in deep learning frameworks. In *32nd USENIX Security Symposium*, 2023.

[19] Bang Di, Jianhua Sun, and Hao Chen. A study of overflow vulnerabilities on GPUs. In *Network and Parallel Computing: 13th IFIP WG 10.3 International Conference*, 2016.

[20] Bang Di, Jianhua Sun, Hao Chen, and Dong Li. Efficient buffer overflow detection on GPU. *IEEE Transactions on Parallel and Distributed Systems*, 32(5), 2021.

[21] Bang Di, Jianhua Sun, Dong Li, Hao Chen, and Zhe Quan. Gmod: A dynamic GPU memory overflow detector. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2018.

[22] Christopher Erb, Mike Collins, and Joseph L. Greathouse. Dynamic buffer overflow detection for GPGPUs. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2017.

[23] Google. Google queue hardening. Available at https://security.googleblog.com/2019/05/queue-hardening-enhancements.html.

[24] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. Microsecond-scale preemption for concurrent GPU-accelerated DNN inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2022.

[25] Ari B Hayes, Fei Hua, Jin Huang, Yanhao Chen, and Eddy Z Zhang. Decoding CUDA binary. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2019.

[26] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *Computer Vision–ECCV 2016*, 2016.

[27] Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. *arXiv preprint arXiv:2009.03300*, 2020.

[28] Sainathan Ganesh Iyer and Anurag Dipakumar Pawar. GPU and CPU accelerated mining of cryptocurrencies and their financial analysis. In *2nd International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud)*, 2018.

[29] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of Large-Scale Multi-Tenant GPU clusters for DNN training workloads. In *USENIX Annual Technical Conference (ATC)*, 2019.

[30] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed DNN training in heterogeneous GPU/CPU clusters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[31] Yonghae Kim, Jaekyu Lee, and Hyesoon Kim. Hardware-based always-on heap memory safety. In *53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020.

[32] Albert Kwon, Udit Dhawan, Jonathan M Smith, Thomas F Knight Jr, and Andre DeHon. Low-fat pointers: Compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. In *Proceedings of the ACM SIGSAC conference on Computer & communications security (CCS)*, 2013.

[33] Jaewon Lee, Yonghae Kim, Jiashen Cao, Euna Kim, Jaekyu Lee, and Hyesoon Kim. Securing GPU via region-based bounds checking. In *Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA)*, 2022.

[34] Tao Liao, Yongjie Zhang, Peter M Kekenes-Huskey, Yuhui Cheng, Anushka Michailova, Andrew D McCulloch, Michael Holst, and J Andrew McCammon. Multicore CPU or GPU-accelerated multiscale modeling for biomolecular complexes. *Computational and Mathematical Biophysics*, 1(2013):164–179, 2013.

[35] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin transformer: Hierarchical vision transformer using shifted windows. In *Proceedings of the IEEE/CVF international conference on computer vision (ICCV)*, 2021.

[36] Frank Luna. *Introduction to 3D Game Programming with DirectX 12*. Mercury Learning and Information, 2016.

[37] John Michalakes and Manish Vachharajani. GPU acceleration of numerical weather prediction. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, 2008.

[38] Andrea Miele. Buffer overflow vulnerabilities in CUDA: a preliminary analysis. *Journal of Computer Virology and Hacking Techniques*, 12:113–120, 2016.

[39] M Miller. Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape. 2019.

[40] NVIDIA. Compute sanitizer. Available at https://docs.nvidia.com/cuda/compute-sanitizer/index.html.

[41] NVIDIA. CUDA C++ programming guide. Available at https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

[42] NVIDIA. NVIDIA Grace Hopper Superchip. Available at https://www.nvidia.com/en-us/data-center/grace-hopper-superchip/.

[43] NVIDIA. NVLink and NVSwitch. Available at https://www.nvidia.com/en-us/data-center/nvlink.

[44] NVIDIA. Parallel thread execution ISA version 8.2. Available at https://docs.nvidia.com/cuda/parallel-thread-execution/index.html.

[45] NVIDIA. Pascal MMU format changes. Available at https://nvidia.github.io/open-gpu-doc/pascal/gp100-mmu-format.pdf.

[46] NVIDIA. Tuning CUDA applications for NVIDIA Ampere GPU architecture. Available at https://docs.nvidia.com/cuda/ampere-tuning-guide/index.html.

[47] Jay H Park, Gyeongchan Yun, M Yi Chang, Nguyen T Nguyen, Seungmin Lee, Jaesik Choi, Sam H Noh, and Young-ri Choi. HetPipe: Enabling large DNN training on (whimpy) heterogeneous GPU clusters through integration of pipelined model parallelism and data parallelism. In *USENIX Annual Technical Conference (ATC)*, 2020.

[48] Sang-Ok Park, Ohmin Kwon, Yonggon Kim, Sang Kil Cha, and Hyunsoo Yoon. Mind control attack: Undermining deep learning with GPU memory exploitation. *Computers & Security*, 102:102115, 2021.

[49] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[50] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 15(1), 2012.

[51] Doaa Abdul-Hakim Shehab and Omar Abdullah Batarfi. RCR for preventing stack smashing attacks bypass stack canaries. In *2017 Computing Conference*, 2017.

[52] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. Nexus: A GPU cluster engine for accelerating DNN-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[53] Dave Shreiner and The Khronos OpenGL ARB Working Group. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1*. Addison-Wesley Professional, 7th edition, 2009.

[54] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[55] Nikko Ström. Scalable distributed dnn training using commodity GPU cloud computing. 2015.

[56] Peng Sun, Wansen Feng, Ruobing Han, Shengen Yan, and Yonggang Wen. Optimizing network performance for distributed DNN training on GPU clusters: Imagenet/Alexnet training in 1.5 minutes. *arXiv preprint arXiv:1902.06855*, 2019.

[57] Mohamed Tarek Ibn Ziad, Sana Damani, Aamer Jaleel, Stephen W Keckler, and Mark Stephenson. cuCatch: A debugging tool for efficiently catching memory safety violations in CUDA applications. In *Proceedings of the ACM on Programming Languages (PLDI)*, 2021.

[58] Zhenkai Zhang, Tyler Allen, Fan Yao, Xing Gao, and Rong Ge. Tunnels for bootlegging: Fully reverse-engineering GPU TLBs for challenging isolation guarantees of NVIDIA MIG. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2023.

[59] Mohamed Tarek Ibn Ziad, Miguel A Arroyo, Evgeny Manzhosov, Ryan Piersma, and Simha Sethumadhavan. No-fat: Architectural support for low overhead memory safety checks. In *ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021.

A GPU List

Table 4 lists all the GPUs on which we have verified the conclusions presented in Section 4.

Table 4: The complete list of tested GPUs.

Architecture	GPU Model
Volta	Tesla V100
Turing	GeForce RTX 2070
Ampere	GeForce RTX 3080
Ampere	A100
Ada Lovelace	GeForce RTX 4090

B ROP Gadgets

An example of the ROP gadget in Section 5 is shown in Listing 6. Each execution of this gadget modifies a weight value, and we execute it repeatedly to modify multiple weights. Specifically, we prepare the stack so that each time the gadget executes, two conditions are met: 1) the address of the first instruction in the gadget is stored at `[R1+0x18]` and is thus loaded into `R20`; 2) the address of the next weight is stored at `[R1+0x38]` and is thus loaded into `R28`. Consequently, each execution of this gadget first modifies a weight value using the `ST` instruction, then updates `R28` to the address of the next weight, and finally returns to the start of the gadget to modify the subsequent weight. If there are a large number of weights to modify and the stack size is insufficient to support the repeated execution of this gadget, we use multiple malicious user requests—and thus multiple ROP attacks—to complete the task.

```
ST.E.64 [R28.64], R4 ;
BSYNC B7 ;
LDL R0, [R1] ;
BMOV.32 B6, R27 ;
LDL R20, [R1+0x18] ;
LDL R21, [R1+0x1c] ;
LDL R2, [R1+0x4] ;
LDL R16, [R1+0x8] ;
...
LDL R19, [R1+0x14] ;
LDL R22, [R1+0x20] ;
...
LDL R29, [R1+0x3c] ;
IADD3 R1, R1, 0x40, RZ ;
BMOV.32 B7, R0 ;
RET.ABS.NODEC R20 0x0 ;
```

Listing 6: An example ROP gadget.

C Shellcode

An example of the shellcode used in the code injection attack is shown in Table 7. It writes the value `0xffffffff` to a range of memory addresses, from `0x7fffdeadbeef` to `0x7fffdeadbeef+0xaaaa`.

D System Specifications

The cloud system used for the experiments in Section 5 is specified in Table 5.

Table 5: Platform details.

CPU	Intel Xeon Silver 4114
Hypervisor	KVM on Ubuntu 20.04
Virtual machine	Ubuntu 20.04
GPU	NVIDIA A100
GPU architecture	Ampere
vGPU version	15.4
vGPU license	vWS
vGPU memory	40GB
CUDA version	12.4

E LLM Attacks

We test the memory corruption attacks, including the code injection attack and the ROP attack, on three LLMs, Flan-T5-Small, Flan-T5-Base, and Flan-T5-Large [5]. We assume a buffer overflow vulnerability in the data pre-processing stage (e.g., for noise removing [6]), similar to the one assumed in Section 5.

Table 6: The LLM inference accuracy (with MMLU [27]) after the weight modification attacks.

DNN Model	Clean acc.	Uncontrolled Weight modification			
		10%	20%	50%	100%
Flan-T5-Small	29.5%	29.2%	28.6%	28.3%	28.2%
Flan-T5-Base	34.2%	33.2%	32.6%	28.9%	29.5%
Flan-T5-Large	42.0%	41.3%	40.5%	39.1%	35.0%

Similar to the attacks on the vision models (cf. Section 5), the code injection attack (with controlled written values) can reduce the accuracy of the LLMs to the same level of random guessing. However, as shown in Table 6, the ROP attack, where the written value is not controlled by the attacker, has a limited effect on the accuracy of the LLMs.

F JOP+ROP Attack

An example of combining JOP and ROP gadgets is shown in Figure 8. Here we assume that `R4` contains the address of the helper gadget, which is used to chain the JOP gadgets (with the ROP gadgets). Every time after executing a JOP gadget, it jumps to the helper gadget, which then pops the address of the next gadget from the stack and returns to it.

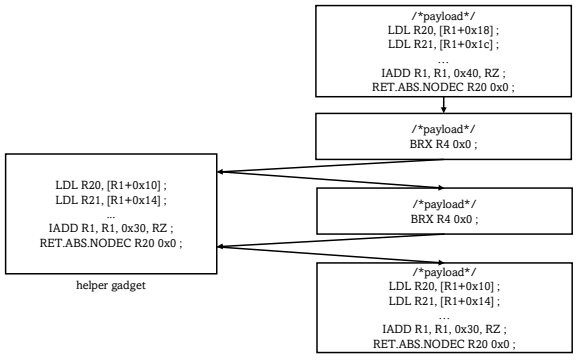

Figure 8: An example of combining JOP and ROP, assuming that `R4` contains the address of the helper gadget.

Table 7: An example shellcode.

```
/* 0000 */   MOV R0, 0xffffffff ;
                                /* 0xffffffff00007802 */
                                /* 0x003fde0000000f00 */
/* 0010 */   MOV R4, 0xdeadbeef ;
                                /* 0xdeafbeef00047802 */
                                /* 0x003fde0000000f00 */
/* 0020 */   MOV R5, 0x7fff ;
                                /* 0x00007fff00057802 */
                                /* 0x003fde0000000f00 */
/* 0030 */   MOV R3, RZ ;
                                /* 0x000000ff00037202 */
                                /* 0x003fde0000000f00 */
/* 0040 */   MOV R6, R3 ;
                                /* 0x0000000300067202 */
                                /* 0x003fde0000000f00 */
/* 0050 */   MOV R3, R6 ;
                                /* 0x0000000600037202 */
                                /* 0x003fde0000000f00 */
/* 0060 */   ISETP.LT.AND P0, PT, R3, 0xaaaa, PT ;
                                /* 0x0000aaaa0300780c */
                                /* 0x003fde0003f01270 */
/* 0070 */   PLOP3.LUT P0, PT, P0, PT, PT, 0x8, 0x0 ;
                                /* 0x000000000000781c */
                                /* 0x003fde000070e170 */
/* 0080 */   @P0 BRA 0x150 ;
                                /* 0x000000c000000947 */
                                /* 0x003fde0003800000 */
/* 0090 */   MOV R6, R3 ;
                                /* 0x0000000300067202 */
                                /* 0x003fde0000000f00 */
/* 00a0 */   SHF.R.S32.HI R7, RZ, 0x1f, R6 ;
                                /* 0x0000001fff077819 */
                                /* 0x003fde0000011406 */
/* 00b0 */   SHF.L.U64.HI R7, R6, 0x2, R7 ;
                                /* 0x0000000206077819 */
                                /* 0x003fde0000010207 */
/* 00c0 */   SHF.L.U32 R6, R6, 0x2, RZ ;
                                /* 0x0000000206067819 */
                                /* 0x003fde00000006ff */
/* 00d0 */   MOV R10, R4 ;
                                /* 0x00000004000a7202 */
                                /* 0x003fde0000000f00 */
/* 00e0 */   MOV R11, R5 ;
                                /* 0x00000005000b7202 */
                                /* 0x003fde0000000f00 */
/* 00f0 */   IADD3 R6, P0, R10, R6, RZ ;
                                /* 0x000000060a067210 */
                                /* 0x003fde0007f1e0ff */
/* 0100 */   IADD3.X R7, R11, R7, RZ, P0, !PT ;
                                /* 0x000000070b077210 */
                                /* 0x003fde00007fe4ff */
/* 0110 */   ST.E [R6.64], R0 ;
                                /* 0x0000000006007985 */
                                /* 0x0033de000c101904 */
/* 0120 */   IADD3 R6, R3, 0x1, RZ ;
                                /* 0x0000000103067810 */
                                /* 0x003fde0007ffe0ff */
/* 0130 */   MOV R7, R6 ;
                                /* 0x0000000600077202 */
                                /* 0x003fde0000000f00 */
/* 0140 */   BRA 0x160 ;
                                /* 0xfffffff0000007947 */
                                /* 0x003fde000383ffff */
/* 0150 */   EXIT ;
                                /* 0x000000000000794d */
                                /* 0x003fde0003800000 */
```