GHost in the Shell: A GPU-to-Host Memory Attack and Its Mitigation

Sihyun Roh, Woohyuk Choi, Jaeyoung Chung, Yoochan Lee, Suhwan Song, and Byoungyoung Lee
Seoul National University
{sihyeonroh, 00cwooh, jjy600901, yoochan10, sshkeb96, byoungyoung}@snu.ac.kr

***Abstract*—Modern heterogeneous computing platforms increasingly unify CPU and GPU address spaces for improved programmability and performance. To this end, recent Linux kernels and NVIDIA GPU drivers adopt Heterogeneous Memory Management (HMM), which allows GPU kernels to directly access host memory. While this simplifies development, it may introduce a critical security risk.**

In this paper, we expose a new attack surface introduced by HMM and present GHOST-ATTACK, the first GPU-originated exploitation technique capable of compromising host process memory. By exploiting memory-safety bugs in GPU kernels or executing attacker-supplied kernels, GHOST-ATTACK enables attackers to bypass Address Space Layout Randomization (ASLR) and hijack control flow in widely used applications such as PyTorch and Chrome.

To counter this threat, we further propose SHELL (Secure HMM Enforcement with LLVM), a practical defense that restores memory isolation between GPU and host in HMM-enabled systems. SHELL statically identifies shared memory regions and enforces fine-grained access control at runtime using the GPU driver's page-fault mechanism. We implement SHELL by modifying Clang/LLVM and the open-source NVIDIA GPU driver. Our evaluation demonstrates that SHELL effectively blocks all variants of GHOST-ATTACK with negligible performance overhead, preserving the security, compatibility, and performance benefits of HMM.

1. Introduction

GPUs have evolved from dedicated graphics processors to powerful parallel computing engines, driving advancements in machine learning, scientific computing, and high-performance applications. General-purpose GPU (GPGPU) programming, which refers to the use of GPUs for tasks beyond traditional graphics rendering, has led to the ubiquitous deployment of GPU-equipped systems. NVIDIA's CUDA platform, in particular, has been a pioneering force in making parallel GPU programming accessible to developers [33].

However, the complexity of GPU programming has driven the need for simplified memory models enabling seamless CPU-GPU interaction. To address this, recent Linux kernels and NVIDIA GPU drivers integrate Heterogeneous Memory Management (HMM) [28], which unifies CPU and GPU address spaces, allowing GPU kernels to directly access host memory [37]. This significantly simplifies data sharing and improves programming efficiency, as developers no longer need to explicitly transfer data between CPU and GPU memory.

With the widespread adoption of GPUs, security researchers have focused on GPU security, primarily targeting vulnerabilities within the GPU memory space. They focused on memory-safety bugs—long standing vulnerabilities in software systems—that can lead to data leaks, corruption, or denial of service within the GPU context [41, 30]. In response to these threats, NVIDIA has introduced tools to detect and debug memory-safety bugs in GPU kernels, such as the Compute Sanitizer [38] and cuda gdb [39]. With the help of these tools, several memory-safety bugs have been reported in NVIDIA's kernel libraries [22, 26]. Moreover, several studies have demonstrated that GPU memory-safety bugs can be exploited to hijack the control flow of GPU kernels and compromise GPU memory [19, 40, 20], leading to data leaks or corruption. However, these previous attacks were limited to the GPU memory and could not affect the host code or memory directly.

This paper studies the security implications of HMM supports in GPU. Specifically, while HMM improves programmability, it is vulnerable to a critical security attack, which we call GHOST-ATTACK. GHOST-ATTACK is the first GPU-to-host memory attack that exploits HMM to compromise the host process. GHOST-ATTACK enables an attacker to exploit GPU memory-safety bugs to directly affect host code and memory, leading to arbitrary code execution on the host. Since HMM is enabled by default in the latest NVIDIA GPU drivers, even CUDA applications not designed to use HMM can still be affected by GHOST-ATTACK [37].

By exploiting memory-safety vulnerabilities in GPU kernels or submitting malicious GPU kernels, GHOST-ATTACK demonstrates how attackers can break Address Space Layout Randomization (ASLR), scan host memory, and then hijack the control flow of host applications. We demonstrate that GHOST-ATTACK can be practically mounted against real-world applications, including PyTorch [12] and Chrome's GPU process [1], using either memory-safety vulnerabilities in GPU kernels or attacker-supplied GPU kernels.

To mitigate this threat, this paper further proposes SHELL (Secure HMM Enforcement with LLVM), the first practical defense that restores memory isolation between GPU and host in HMM-enabled systems. SHELL achieves this with a two-phase approach: it (1) statically identifies legitimate shared memory regions passed to the GPU, and (2) enforces fine-

grained runtime access control by leveraging existing page-fault mechanisms. Importantly, SHELL requires no manual changes to GPU applications or hardware. All protections are transparently applied using a compiler-based instrumentation. Moreover, it introduces minimal overhead by reusing existing HMM mechanisms and enforcing access control only during page faults, further optimized by 64KB-aligned allocations to avoid sub-page sharing.

We evaluate SHELL against the ERA5 dataset [3] processing application developed by NVIDIA [4] to demonstrate the effectiveness of HMM, and show that SHELL introduces negligible overhead on end-to-end performance. Moreover, SHELL uses jemalloc [5]-based memory pools to improve memory allocation speed regard to ML/AI workloads, which imposes a high memory pressure on the host memory. Our evaluation shows that SHELL effectively blocks all variants of GHOST-ATTACK while maintaining full compatibility with existing CUDA applications. Moreover, SHELL imposes negligible overhead: on both microbenchmarks and a real-world HMM-enabled CUDA application, SHELL preserves performance and even improves memory allocation speed through the use of jemalloc-based memory pools.

Although HMM is not yet widely adopted due to its recent introduction, major GPU vendors, including NVIDIA and AMD, are all moving towards supporting HMM in their GPU drivers and hardware. NVIDIA has already enabled HMM by default in its latest GPU drivers, and AMD has announced plans to support HMM in future releases [17, 37]. This trend is driven by the need for more ease of memory management and programmability in heterogeneous computing environments, especially in the context of machine learning and AI applications, which imposes high memory pressure on the host memory. Our work highlights the urgent need to address the security implications of HMM, which lacks security considerations in its usecases, and provides a practical solution to mitigate the risks associated with HMM-enabled systems.

The rest of this paper is organized as follows. §2 provides backgrounds related to GPU software stacks and HMM support. §3 presents GHOST-ATTACK, which gains a host process's privilege by compromising GPU kernels or supplying attacker-controlled GPU kernels. §4 proposes SHELL, which is an effective mitigation against GHOST-ATTACK. §5 describes the implementations of SHELL. §6 evaluates the attack effectiveness of GHOST-ATTACK as well as security and performance effectiveness of SHELL. §7 discusses the alternative mitigations and future work of this paper, and §9 concludes the paper.

Open Science Policy. In support of open science and reproducible research, we publicly released all our artifacts, including proof-of-concept code for GHOST-ATTACK and the implementation of SHELL at our public repository [18].

Responsible Disclosure. We first identified the GHOST-ATTACK and security flaws in CUDA runtime on April 17, 2025, and promptly reported them to NVIDIA PSIRT on June 2, 2025. Our Initial submission included a high-level description of how HMM and CUDA runtime can be exploited by an attacker, followed by a proof-of-concept code that demonstrates the attack on a vulnerable CUDA kernel, and a prototype of SHELL.

2. Background

In this section, we provide background information on the GPU and Heterogeneous Memory Management (HMM) [28]. First, we review the GPU software stack, which includes the host program, the CUDA runtime library, the GPU driver, and the GPU kernel (§2.1). Next, we introduce the GPU programming model and its workflow, which is essential for understanding how GPU kernels are executed (§2.2). We then explain the GPU memory management, including the traditional GPU memory management, the UVM, and the HMM, which is a new memory management model (§2.3).

2.1. GPU Software Stack

The GPU software stack is composed of several components that operate on both the host and GPU sides. The host-side components run on the CPU, and manage GPU's execution, memory, and kernel launches, while the GPU-side component runs on the GPU, and performs the actual computation.

Host Program (Host-Side). The host program is a user-space application developed by third-party developers who wish to leverage GPU acceleration for their computations. Common examples of the host program include deep learning or machine learning frameworks such as PyTorch [12] and TensorFlow [13]. These include various GPU kernels and are built with `libcuda-rt`, both of which are explained next.

CUDA Runtime Library (Host-Side). The CUDA runtime library (`libcuda-rt`) [33], provided by NVIDIA, serves as an abstraction layer between the host program and the GPU driver. It exposes high-level APIs to the host program, which are then forwarded to the GPU driver. Leveraging this layer, host programs can simplify its GPU programming—such as managing GPU memory, GPU kernel launch, and other GPU operations. Basically this `libcuda-rt` is a shared library, so the host program dynamically links the library when executed.

GPU Driver (Host-Side). The GPU driver provides low-level interfaces and system-level control over the GPU hardware. It provides low-level APIs utilized by `libcuda-rt` for GPU resource management, including memory allocation, kernel preparation, and execution. Furthermore, it coordinates the transfer the GPU kernels to the GPU memory and the execution of those.

GPU Kernel (GPU-Side). GPU kernels are functions executed on the GPU hardware. NVIDIA provides a set of optimized, closed-source GPU kernels (e.g., cuBLAS [32], cuDNN [34], and cuFFT [35]). These kernels perform specific computational tasks, such as linear algebra, image processing, and machine learning. NVIDIA provides highly optimized prebuilt GPU kernels so that host program developers can benefit from vendor-tuned performance. It is

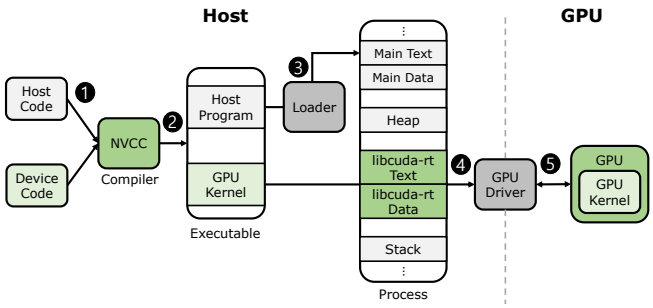


Figure 1: Overview of how NVIDIA GPU kernels are compiled and executed.

```
1   int main() {
2       float *h_data;
3       h_data = (float *)malloc(SIZE);
4
5   -   float *d_data;
6   -   cudaMalloc((void**)&d_data, SIZE);
7   -   cudaMemcpy(d_data, h_data, SIZE, cudaMemcpyHostToDevice);
8   -   cuda_kernel<<<blockDim, threadDim>>>(d_data);
9   -   cudaMemcpy(h_data, d_data, SIZE, cudaMemcpyDeviceToHost);
10
11  +   cuda_kernel<<<blockDim, threadDim>>>(h_data);
12
13      ...
14  }
```

Figure 2: A host program which uses CUDA, shown with diff-style comparison before and after adopting HMM. Red-lines indicate code written for pre-HMM version, while green-lines indicate new code lines written for using HMM.

also possible for host program developers to write their own GPU kernels to support specialized computations that are not covered by NVIDIA-supplied GPU kernels.

2.2. GPU Programming

Modern GPUs execute specialized programs to accelerate parallel computations. Two key abstractions for these programs are GPU kernels and shaders. GPU kernels are commonly used in general-purpose GPU programming frameworks like CUDA [33] or OpenCL [9], and shaders are used in graphics APIs such as OpenGL [10] and Direct3D [2]. Although both run on the same underlying hardware, they differ in their programming model, execution context, and intended use cases. In this paper, we focus on GPU kernels, particularly NVIDIA CUDA kernels, which are designed for general-purpose computations and can be executed on a wide range of data types and structures.

The overall process of compilation and execution of the GPU kernel is illustrated in Figure 1. ❶ The programmer writes the host code and the device code. Host code is a C/C++ code that runs on the CPU as a user-level process, responsible for managing the GPU resources, transferring data between the CPU and GPU. Device code is a CUDA C/C++ code that compiled to a device-executable format (i.e., PTX [11] or SASS), and executed on the GPU. In NVIDIA, the device code running on the GPU is called a GPU kernel, which is a function that can be executed in parallel by multiple threads. Both host and device code are compiled by the NVIDIA CUDA compiler (NVCC [7]) into a single executable. ❷ The executable consists of two parts: the CPU binary and the GPU binary (fatbin). The CPU binary are composed of the code and data that are executed and accessed by the host. The GPU binary is a device-executable format (i.e., PTX or SASS) that is executed on the GPU. ❸ When a user runs the executable, the loader loads the CPU binary into the host process and begins execution. Once executed, the host code calls the APIs of `libcuda-rt`, which is dynamically linked to the host program. ❹ When the host code calls GPU kernels, it invokes the `cudaLaunchKernel` from `libcuda-rt`, which transfers the GPU binary to the GPU memory and prepares the GPU kernel for execution with support of the GPU driver. ❺ The interaction between the host and the GPU is kept being managed by `libcuda-rt` and the GPU driver, ensuring seamless communication and efficient execution of GPU kernels.

2.3. GPU Memory Management

Traditional GPU Memory Management. Traditionally, the host (CPU) and device (GPU) each had separate address spaces, and every host-to-device transfer had to be manually managed by the host program. In Figure 2, the code snippets with the red-colored lines show the workflow of this legacy GPU memory management. Specifically, a traditional host program using CUDA performs the following steps for memory management.

- **Host Memory Allocation:** The host program allocates the host memory using standard C/C++ memory allocation functions (e.g., `malloc()` or `new`) (line 2-3).
- **GPU Memory Allocation:** The host program reserves GPU memory using `libcuda-rt` APIs (i.e., `cudaMalloc()`). This API returns a device pointer (i.e., `d_data`). Since this pointer points to the GPU memory, it cannot be used by the host code (line 5-6).
- **Kernel Launch:** The host program launches the GPU kernel with passing the device pointer (`d_data`) to the kernel function (line 8), and this pointer now can be used in the GPU kernel to access the GPU memory.
- **Kernel Execution:** The GPU kernel is executed on the GPU, and it operates on the data in the GPU memory.
- **Data Retrieval:** After the GPU kernel finishes the execution, the host program copies the data from the GPU memory back to the host memory using `libcuda-rt` APIs (i.e., `cudaMemcpy()`) (line 9).

As described above, CUDA memory management involves complicated steps for developers, requiring explicit memory allocation, data transfers, and synchronization between the host and device.

GPU Memory Management with UVM. Unified Virtual Memory (UVM) in CUDA is a memory management model that allows the host and GPU to share a single unified address space. With UVM, the host program allocates memory accessible to both the host and GPU using the explicit API call `cudaMallocManaged()`, while all other memory regions

remain unshared between the host and GPU. The pointer returned by `cudaMallocManaged()` can be directly used by both host and GPU code. The data transfers between the host and GPU and synchronization are automatically managed by the GPU driver and CUDA runtime, thus reducing the complexity of memory management for developers.

GPU Memory Management with HMM. Heterogeneous Memory Management (HMM) [28], which is a kernel framework in Linux, allows a device (e.g., GPU, FPGA, and NIC) to access pages allocated in the host memory as if it were part of the device's address space. HMM is supported by NVIDIA GPUs and their drivers, enabling a GPU kernel to directly access host memory. HMM is designed to be more seamless and efficient than UVM, making it superior to UVM in terms of programmability. It simplifies host code by allowing standard memory allocations (e.g., `malloc()`, `new`, or stack allocation) to be directly accessible to the GPU kernel. This offers the advantage in programmability compared to UVM, which requires explicit allocation through `cudaMallocManaged()`. Given these advantages, NVIDIA recommends developers to use HMM for ease of programming [37]. Figure 2 shows the HMM's simplified memory management workflow compared to the traditional GPU memory management. With HMM, the code highlighted in red is no longer needed, and line 9 is replaced with a simple argument passing of the host pointer (i.e., `h_data`) as in line 11. NVIDIA GPU architectures now enable the HMM by default since the open-source NVIDIA GPU driver version r535_00+ with CUDA Toolkit version 12.2+ (which runs Linux kernel 6.1.24+, 6.2.11+, or 6.3+) [37].

3. GHOST-ATTACK

In this section, we propose GHOST-ATTACK, the first GPU-to-host attack that compromises host memory integrity through Heterogeneous Memory Management (HMM). Specifically, we begin by showing how host-side ASLR can be broken under HMM (§3.1), then explain how an attacker can hijack the host program's control flow (§3.2). Next, we discuss the security consequences of GHOST-ATTACK (§3.3). Finally, we present two realistic attack scenarios that demonstrate the practical feasibility of GHOST-ATTACK (§3.4).

Attack Model. We assume a heterogeneous system with HMM enabled. The victim's host program is a user process, such as PyTorch or TensorFlow, that dynamically links `libcuda-rt` to utilize GPU acceleration.

We consider two realistic attack vectors: (i) exploiting a memory-safety vulnerability in a GPU kernel using crafted inputs, and (ii) executing attacker-supplied GPU code in the context of the host (i.e., via WebGPU [42] or HIPscript-like APIs [27]). In both cases, the attacker first gains the privilege of the GPU kernel context. Based on this compromised GPU kernel, the attacker's goal is to compromise the control flow or data integrity of the host process.

Attack Overview. Although a compromised or malicious GPU kernel can directly access host memory,

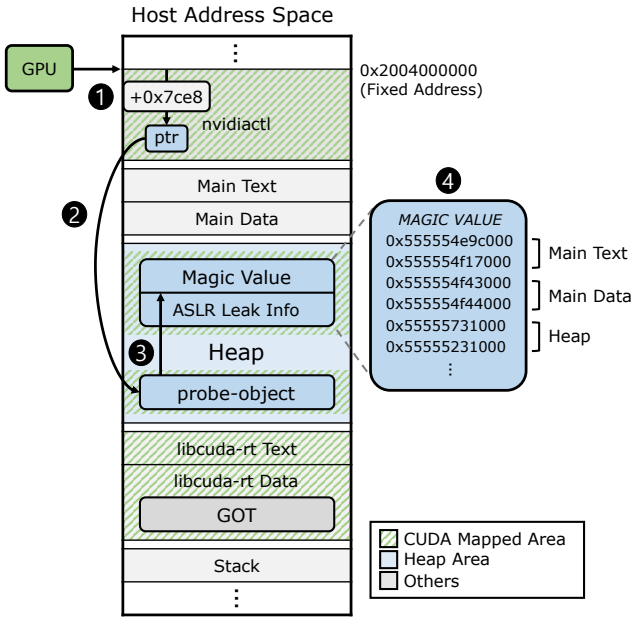


Figure 3: Memory snapshot of a host process, showing how GHOST-ATTACK breaks ASLR.

GHOST-ATTACK still requires to complete the following two attacking tasks to successfully gain the privilege of the host process. First, GHOST-ATTACK needs to bypass ASLR. The attacker must first learn the layout of the host process's address space, which is randomized for security. Without this information, any attempt to overwrite critical data structures would likely crash the host process or fail silently. Second, GHOST-ATTACK needs to hijack the control flow to gain the host process's privilege. Specifically, even after bypassing ASLR, the attacker must find a writable control-relevant data (such as a return address or a function pointer), which can be used to redirect the execution flow.

The rest of this section describes how GHOST-ATTACK carries out these attacks by exploiting one practical and broadly applicable attack vector: the CUDA runtime library (`libcuda-rt`). While GHOST-ATTACK can leverage multiple attack vectors to corrupt host control flow, we focus here on an attack that targets `libcuda-rt` because it is dynamically linked by all CUDA-using host programs and thus is always mapped into the host process's address space. This choice makes the attack broadly applicable to all CUDA-enabled applications. Alternatively, one may define a targeted attack vector tailored to the characteristics of a specific target system instead of `libcuda-rt`, but general attack strategy would be similar to this `libcuda-rt` attack

3.1. Breaking ASLR with `libcuda-rt`

GHOST-ATTACK breaks ASLR by exploiting multiple design decisions in `libcuda-rt` that inadvertently exposes critical memory layout information to the GPU. This attack is based on the following three key characteristics of the host

```
1  // GPU kernel to break ASLR and leak addresses
2
3  __global__
4  void aslr_break_kernel() {
5      uint64_t nvidiactl_base = 0x200400000ULL;
6      uint64_t offset  = 0x7ce8;
7      uint64_t *probe_object
8                      = *(uint64_t **)(nvidiactl_base + offset);
9
10     uint64_t i = 0;
11     uint64_t *leak_object;
12
13     // find the object that contains the magic values
14     // 0x200000000 and 0x300200000
15     while(true) {
16         if(*(probe_object - i) == 0x200000000ULL) {
17             if(*(probe_object - i + 1) == 0x300200000ULL) {
18                 leak_object = probe_object - i;
19                 break;
20             }
21         }
22         i++;
23     }
24
25     // read objects until it leaks the last mapping information
26     for(i = 0; *(leak_object + i) != 0xffffffffff600000; ++i) {
27         printf("leaked address: %lx\n", *(leak_object + i));
28     }
29 }
```

Figure 4: Pseudo GPU kernel code to break ASLR. The code is translated into pseudo C-style for readability.

process's memory, all of which are related to the runtime behavior of `libcuda-rt`.

- **Fixed Mapping for nvidiactl device** When a GPU kernel is launched, `libcuda-rt` maps the `nvidiactl` device into the host process's address space at a fixed virtual address. This region is used for communication between the host and GPU. Crucially, `libcuda-rt` maps this region to the same virtual address across executions. This behavior violates a fundamental principle of ASLR, as it provides a reliable anchor to begin the search of other host-memory regions' addresses.
- **A Heap Pointer placed at the Fixed Offset.** An attacker can locate the heap region by reading a pointer stored at a fixed offset within the `nvidiactl`-mapped region. Our analysis confirms that this pointer consistently references the host process's heap, enabling computation of other heap addresses. Specifically, at the fixed offset 0x7ce8 from the `nvidiactl`-mapped region, a 64-bit pointer always points to a `probe-object`, which is located in the process's heap (the same heap used by the host for all allocations).
- **Host Address Layout Information in Heap** Analyzing the data within the heap, we found that a certain data structure, which stores the base and end addresses of all memory sections in the host process, including text, data, heap, stack, and shared libraries. Our further analysis identified that this memory layout structure in the heap is initialized and managed by `libcuda-rt` to facilitate low-level resource accesses when operating with the GPU driver. By parsing this structure, the attacker can reconstruct the full memory layout of the host process, effectively bypassing ASLR.

Together, these memory characteristics of the host process allow a compromised or malicious GPU kernel to deterministically locate and leak the memory layout of the host process with no guesswork or brute force. The combination of a fixed anchor, stale heap pointer exposure, and rich address information metadata makes `libcuda-rt` an ideal target for ASLR bypass. Even in the absence of `libcuda-rt`, a host process generally passes pointers (holding a host address) to GPU kernels as arguments in HMM scenarios, resulting in the host address leakage. This makes the attack broadly applicable to any host program (beyond `libcuda-rt`) that employs HMM.

Workflow of ASLR Bypass. In order to clearly illustrate how this ASLR bypass can be carried out by the GPU kernel, we provide details of the attack flow. The memory snapshot throughout the attacking process is shown in Figure 3 and C-like attack pseudocode is listed in Figure 4. First, from the fixed `nvidiactl`-mapped address, the GPU kernel first determines the address of a pointer (i.e., 0x200407ce8), which is located at the fixed offset from the fixed `nvidiactl`-mapped region (❶ in Figure 3 and line 5-6 in Figure 4.). Then it dereferences this pointer, which yields the pointer of the `probe-object` (❷ and line 7-8). Next, it scans below the `probe-object` address to identify a `leak-object` that begins with a 16-byte magic value (❸ and line 15-23). Finally, it extracts the host memory layout information (❹ and line 26-28).

```
1  // GPU kernel to overwrite a return address on the stack
2
3  __device__
4  void return_address_overwrite_payload() {
5      // Break ASLR, and leak the libcuda base
6      size_t libcuda_base = find_libcuda_base();
7      // return address to overwrite
8      size_t return_addr  = libcuda_base + 0x25f701;
9
10     // Scan the stack to locate the target return address
11     size_t target_addr = scan_stack(return_addr);
12
13     // Scans host address space to find the gadget
14     uint64_t gadget_addr = find_gadget_addr();
15     // Overwrite the target return address with the gadget address
16     *(uint64_t *)target_addr = gadget_addr;
17 }
```

(a) GPU kernel code that overwrites return address.

```
1  // GPU kernel to overwrite libcuda GOT entry
2
3  #define GADGET_OFFSET (0xXXXXXX)
4  #define FREE_OFFSET (0x25f701)
5
6  __device__
7  void GOT_overwrite_payload() {
8      // Break ASLR, and leak the libc & libcuda base
9      size_t libc_base = find_libc_baes();
10     size_t libcuda_base = find_libcuda_base();
11
12     // Set Gadget address to overwrite (gadget_addr)
13     size_t gadget_addr = libc_base + GADGET_OFFSET;
14     // Set GOT entry to be overwritten (target_addr)
15     size_t target_addr = libcuda_base + FREE_OFFSET;
16
17     *(uint64_t *)target_addr = gadget_addr;
18 }
```

(b) GPU kernel code that overwrites GOT entry.

Figure 5: GPU kernel code to hijack control flow of the host program.

3.2. Hijacking Host Control-Flow

After bypassing ASLR, now we describe how the GPU kernel can leverage HMM to hijack the host process's control flow by overwriting critical data structures, such as return addresses and function pointers. We first describe the attack with overwriting return addresses, and then show how to overwrite function pointers in the Global Offset Table (GOT) to achieve arbitrary code execution in the host context.

Overwriting Return Addresses of `libcuda-rt`. First, we explain how a compromised GPU kernel can overwrite return addresses in the host process (shown in Figure 5a). After defeating ASLR and learning the host's memory layout, the GPU kernel can locate the host's stack region. However, it introduces a challenge in locating the valid return address of the host program, because randomly overwriting stack data risks crashing the host, which is undesirable for the attacker. To address this, we leverage the fact that the host program waits for the GPU kernel to finish its execution via `cudaDeviceSynchronize()`. When `cudaDeviceSynchronize()` is called, it calls a sequence of `libcuda-rt` functions that eventually invoke a blocking loop that waits for the GPU kernel to finish execution. The `libcuda-rt` function that runs the blocking loop stores the return address, whose value is the address of `0x25f701` offset from the base address of `libcuda-rt`, in the host stack. From the GPU kernel, we can scan the stack region of the host process to find this unique return address value (line 6-11 of Figure 5a). Once the GPU kernel locates the target return address, attacker can overwrite it (and any subsequent stack slots) with attacker-controlled values and gadget addresses taken from the host program's executable section. When the blocking loop in `libcuda-rt` is finished, it returns to the attacker-controlled address instead of the original return address.

Overwriting GOT of `libcuda-rt`. The second method to hijack the host's control flow is to overwrite function pointers in the Global Offset Table (GOT) of `libcuda-rt`. `libcuda-rt` is built with partial RELRO, which leaves the GOT writable. Thus, the compromised GPU kernel can overwrite GOT to hijack the control flow as follows (shown in Figure 5b). First, the GPU kernel leverages HMM to write directly to the GOT entry of a chosen function (e.g., `free()`). Next, once `libcuda-rt` invokes the function which relies on the overwritten GOT entry, the control-flow would be redirected the entry to the attacker-overwritten address, achieving arbitrary code execution in the host context. Figure 5b shows the example C-style pseudocode of the attack, where the GPU kernel overwrites the GOT entry of `free()` to redirect execution to an attacker-controlled address. It first breaks ASLR, and then identifies the base address of `libcuda-rt` and the `libc` (line 9-10). Then, it calculates the gadget address from the `libc` base, and the target address (i.e., address of `free()` in `libcuda-rt` GOT entry) to overwrite. Finally, it overwrites the GOT entry of `free()` in `libcuda-rt` GOT entry with the attacker-controlled gadget address (line 17).

3.3. Security Consequences

Traditionally, GPU kernels are considered less privileged than the host process, with limited capabilities and no direct access to host memory. However, GHOST-ATTACK enables the GPU kernel to compromise the host memory, leading to severe security consequences. First, it breaks host ASLR. Second, it can hijack the host's control flow and achieve arbitrary host code execution using only a GPU memory bug. Host code execution is significantly more dangerous than GPU kernel-only execution because the host can access sensitive data and privileged system resources. Thus, GHOST-ATTACK can lead to severe security consequences.

Host Privilege Escalation. Once the attacker achieves host code execution, they can execute a chain of system calls to exploit vulnerabilities in the host kernel. For example, CUDA-enabled applications often have direct access to GPU driver APIs (e.g., `ioctl()`) even when running in a sandboxed environment. If a vulnerability exists in the GPU driver, an attacker who controls the host can use that functionality to gain kernel-level privileges.

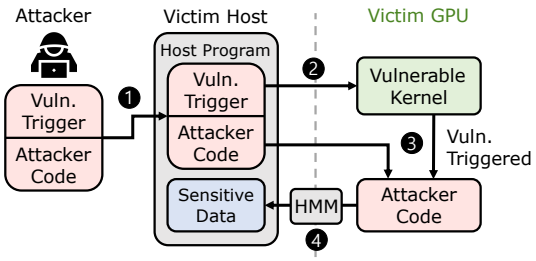
Exfiltration of Sensitive Data. Host code execution allows the attacker to read sensitive data from the host process. Whereas GPU memory corruption within a GPU kernel is limited to data present in GPU memory during that GPU kernel's execution, host code execution lets the attacker read and copy arbitrary data from the host process's memory. It includes data not mapped into GPU memory and data beyond the lifetime of the compromised GPU kernel. For example, in a cloud ML service (e.g., ChatGPT, Gemini, etc.), persistent control of the service could allow an attacker to exfiltrate other clients' inputs (images, text) and model data which are otherwise out of reach for compromised GPU kernels. This capability is especially dangerous in multi-tenant deployments, where sensitive data from different users may be co-located in the same host process.

3.4. Real-world Attack Scenarios

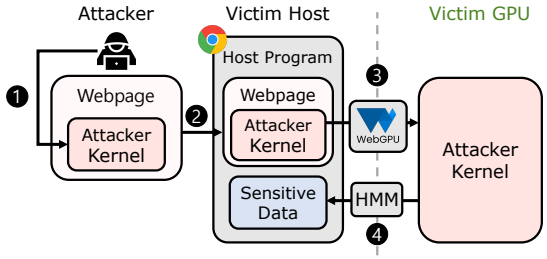
We present two real-world attack scenarios that demonstrate how a GPU kernel, which is compromised or attacker-supplied, can break ASLR and hijack control flow in the host process through HMM. Both attacks first defeat ASLR by exploiting design issues in the CUDA runtime library and then overwrite critical control data to redirect execution to attacker-controlled code. We confirmed that both attacks successfully changed the host instruction pointer to an attacker-supplied address.

3.4.1. Exploiting Vulnerable GPU Kernels in PyTorch.

This attack shows how to exploit the popular deep learning framework, PyTorch. This attack is built on two assumptions: (1) PyTorch uses a GPU kernel that contains a memory-safety vulnerability, such as a buffer overflow, and (2) the attacker is a client of an inference service implemented with PyTorch. These assumptions reflect real-world deployments.



(a) Scenario 1: Exploiting a vulnerable GPU kernel.



(b) Scenario 2: Exploiting an attacker-supplied GPU kernel.

Figure 6: Attack scenarios that demonstrate how GHOST-ATTACK obtain host process's control-flow through HMM.

PyTorch relies on precompiled GPU kernels for performance, and user-supplied input tensors are routinely offloaded to the GPU for inference. Figure 6a illustrates the attack on vulnerable GPU kernels. To exploit this setting, ❶ attacker should craft an input that includes two components: (1) data that triggers the vulnerability in the GPU kernel, and (2) a payload containing attacker code written in GPU ISA (i.e., SASS). ❷ When this input is processed, the vulnerable GPU kernel triggers the buffer overflow, allowing us to overwrite a control-flow relevant data structure, such as a function pointer or a return address in the GPU memory. ❸ Vulnerability trigger overwrites this data with the address of our injected payload. It results in the GPU kernel executing the attacker's code, transferring control to the attacker-controlled GPU code. ❹ Attacker's payload then performs a two-stage attack. First, it breaks ASLR by exploiting design flaws in `libcuda-rt`, as described in §3.1. Then, using the leaked address layout information, it overwrites a return address stored by `cudaDeviceSynchronize()` in the host stack, or a `libcuda-rt`'s GOT entry, with an attacker's gadget address. Once the corresponding function is invoked on the host, control flow is hijacked, demonstrating that a memory bug in a GPU kernel can be escalated to full code execution on the host.

3.4.2. Exploiting Attacker-Supplied GPU Kernels. This attack demonstrates that even without kernel vulnerabilities, an attacker can compromise the host process if it allows execution of attacker-supplied GPU code. Although the current remote GPU execution APIs (i.e., WebGPU / WebNN in Chrome) [42, 15] do not support NVIDIA CUDA, we assume that future APIs will allow execution of GPU kernels written in CUDA. If attacker-supplied GPU kernel is executed in the victim's machine, the attacker GPU kernel can leverage HMM to directly read and write the victim's host memory, bypassing traditional isolation guarantees. This scenario is illustrated in Figure 6b. ❶ The attacker uploads a malicious GPU kernel to a service (i.e., a website or cloud GPU job API), and waits for the victim to load it. ❷ The victim's host program receives the attacker-supplied kernel, and ❸ loads it into the GPU for execution with support of underlying GPU support framework (i.e., WebGPU with CUDA). ❹ The GPU kernel uses HMM to access host memory, scanning for critical data structures such as return addresses, function pointers, or GOT entries in host programs, and overwriting them with attacker-controlled values.

4. Mitigation: SHELL

4.1. Problem Analysis and Design Overview

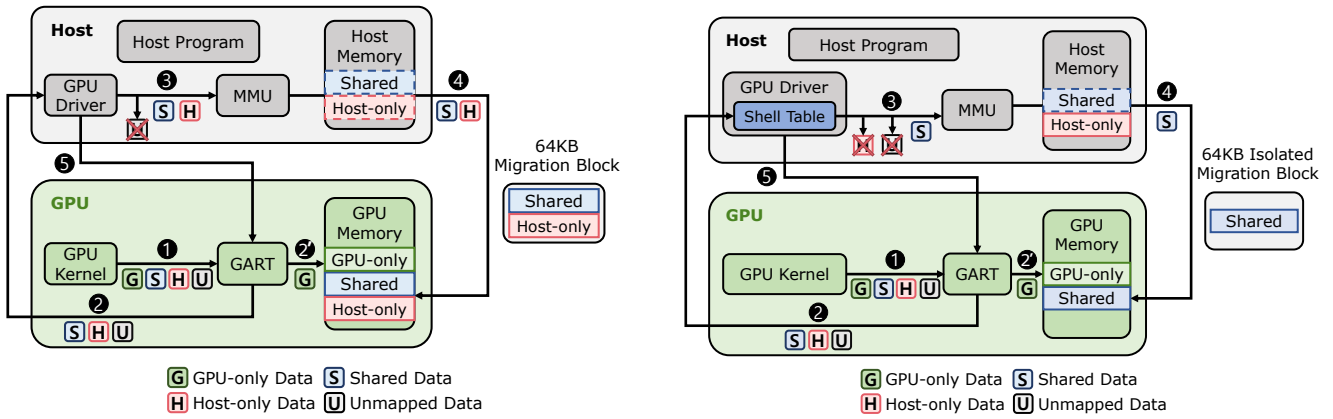
Problem Analysis. The key security problem of HMM is that the GPU kernel has the same level of access to host memory as the host program. This means that the GPU kernel can access more of the host's virtual address space than is actually necessary for correct execution of the GPU kernel. However, this excessive exposure is unnecessary, as the GPU kernel do not need the same level of access to host memory as the host program itself. More precisely, the host address space consists of the following three categories of memory.

- **Host-only data** is allocated in the host memory and accessed only by the CPU (e.g., private host buffers or host control-flow data).
- **GPU-only data** is allocated in the GPU memory and accessed solely within GPU memory (e.g., via `cudaMalloc()`).
- **Shared data** is allocated in the host memory but intentionally shared with the GPU (e.g., passed as kernel arguments).

HMM's security flaw is that it does not enforce any boundaries among these categories. Specifically, GPU kernels should only access `GPU-only data` and `shared data`. The `host-only data` should never be accessible to GPU kernels. If allowed, it breaks the security boundary between CPU and GPU, as we have demonstrated in §3.

Design Overview. To address this problem, we propose SHELL (Secure HMM Enforcement with LLVM), a lightweight system that enforces access control over GPU-initiated memory accesses. Specifically, SHELL aims at preventing GPU kernels from accessing to `host-only data`. As such, SHELL ensures that GPU kernels only access `shared data` and `GPU-only data`.

SHELL enforces memory isolation through a two-phase design: (1) Static phase, which identifies legitimate regions of `shared data` using compiler instrumentation; and (2) Runtime phase, which enforces access control in a GPU driver to validate memory accesses. Static phase is done by instrumenting the host code with Clang/LLVM, designed to be compatible with existing GPU programming practices,



(a) HMM-enabled systems before employing SHELL

(b) HMM-enabled systems after employing SHELL

Figure 7: Access control enforcement in HMM-enabled systems before and after employing SHELL. The left shows the baseline system, where the GPU kernel can access all host memory due to insufficient isolation. The right shows the system after applying SHELL, which modifies the GPU driver's page fault handler and enforces isolated migration blocks to confine shared data, ensuring fine-grained access control.

requiring only a slight modification (i.e., customized flag in the `mmap()` for `shared data` allocation) to the host code and no changes to the GPU kernel code. Runtime phase is implemented in the NVIDIA GPU driver, which intercepts GPU-triggered page faults and checks the faulting address against the precomputed whitelist of `shared data` regions, which is maintained in the SHELL table and updated by the instrumented host code.

4.2. Static Instrumentation for Identifying Shared Data

To enforce secure and precise access control, SHELL must accurately identify memory regions of `shared data`, which are intentionally shared between the host and GPU. A key observation is that all the address information of `shared data` should be handed over to the GPU kernel through CUDA APIs. This is because `shared data` regions are allocated by the host, and thus GPU kernels can only access them if the host explicitly provides their address information. We analyzed CUDA APIs, and found that `cudaLaunchKernel()` is the only API used to pass the shared data address.

SHELL leverages this insight to instrument host programs to identify `shared data`. Using Clang/LLVM, we implement a pass that instruments all `cudaLaunchKernel()` invocations in the host code. These instrumented code first collects all information regarding `shared data` regions, including its base address and size. Then this information is sent to the GPU driver to update SHELL table, a per-kernel whitelist of authorized GPU-accessible regions.

Importantly, these `shared data` regions are not inferred from GPU kernel's memory accesses but are explicitly derived from host-side code. This approach eliminates accidentally identifying `shared data` regions from compromised or attacker-provided GPU kernels.

4.3. Runtime Enforcement with Two-Tiered Access Control

SHELL enforces its access control policy at runtime using light-weight yet robust two-tiered access control mechanisms, which integrates coarse-grained page fault handling with fine-grained allocation constraints. These two mechanisms work together to ensure that GPU kernels can only access `shared data` and `GPU-only data`, while preventing access to `host-only data`.

4.3.1. Coarse-Grained Enforcement via Page Faults.

We built SHELL's coarse-grained access control on top of the existing Heterogeneous Memory Management (HMM) support in the NVIDIA GPU driver. This section first explains how original HMM works on page migration and then describes how SHELL extends it to enforce access control over host memory using page faults.

HMM's Page Migration for Shared Data. As shown in Figure 7a, when a GPU kernel accesses host memory under HMM, there can be the following four cases: (i) accessing the page of `gpu-only data` (Case G), (ii) accessing the page of `shared data` (Case S), (iii) accessing the page of host-only data (Case H), and (iv) accessing an unmapped page by the host (Case U). ❶ In all these four cases, the GPU kernel accesses with a virtual address, and the address is delivered to the Graphics Address Remapping Table (GART). The GART translates the virtual address used by the GPU kernel to the physical address in the GPU memory, similar to the MMU in the CPU. ❷' If the address is mapped to the GPU memory (G), the GART successfully translates the address, and the GPU kernel can access the page in the GPU memory. ❷ If the address is not mapped to the GPU memory (S, H, or U), the GART raises a page fault, which is then delivered to the GPU driver running in the host kernel space. ❸ In case of unmapped page (U), the GPU driver denies

the page migration, and the GPU kernel cannot proceed. In case of `host-only data` (H) and `shared data` (S), the GPU driver serves the page fault by migrating the page from the host memory to the GPU memory. ❹ There is no isolation between `shared data` and `host-only data`. Thus, when the pages are migrated, they contain both `shared data` and `host-only data` without any separation, causing both types of data to reside together on the same page during the migration (S and H). ❺ After the page migration, the GPU driver updates the GPU page table in the GART, allowing the GPU kernel to access the migrated page in the GPU memory (S and H).

Enforcing Coarse-Grained Access Control. Figure 7b illustrates how SHELL extends the HMM's page migration mechanism to enforce access control over host memory. ❶-❷ The GPU kernel under SHELL still accesses the memory using a virtual address, and GART translates the address to the physical address in the GPU memory in case of `gpu-only data` (G), or raises a page fault in case of `shared data` (S), `host-only data` (H), and unmapped page (U). ❸ In order to prevent the `host-only data` from being migrated, SHELL only migrates the page for the case of accessing `shared data` (S), and denies to migrate for cases H and U. ❹ In the case of S, SHELL migrates the pages from the host memory to the GPU memory, ensuring that serving pages contain only `shared data` and not `host-only data` with fine-grained enforcement mechanism, which we will explain in the next subsection. ❺ After the page migration, the GPU driver updates the GPU page table in the GART, allowing the GPU kernel to access the migrated page in the GPU memory (S).

4.3.2. Fine-Grained Enforcement using Isolated Block. While this GPU page-fault based coarse-grained access control successfully prevents the block migration of majority cases, we found that this design alone has sub-page security issue [16]. Specifically, the NVIDIA GPU drivers handle GPU page fault in 64KB granularity, meaning that the faulting address of low 16-bits are masked out. As a result, if `host-only data` and `shared data` are placed in the same 64KB block region, the page fault due to accessing `host-only data` would be allowed by SHELL and thus migrated.

In order to address this sub-page security issue, SHELL further enforces fine-grained access control, ensuring that all `shared data` are placed in its isolated migration block. As a result, `shared data` and `host-only data` are not placed together in the same 64KB of migration block, thus preventing sub-page vulnerabilities. To this end, SHELL first performs the static analysis to identify all variables used as `shared data`. Specifically, it performs the backward use-def analysis starting from the pointer argument of `cudaLaunchKernel()`.

Depending on allocation types, SHELL ensures to place `shared data variables` in the isolated page as follows (illustrated in Figure 8). (i) To secure global variables (i.e., `Main Data` section), SHELL maintains additional `Shared Main Data` region, which is aligned as 64KB

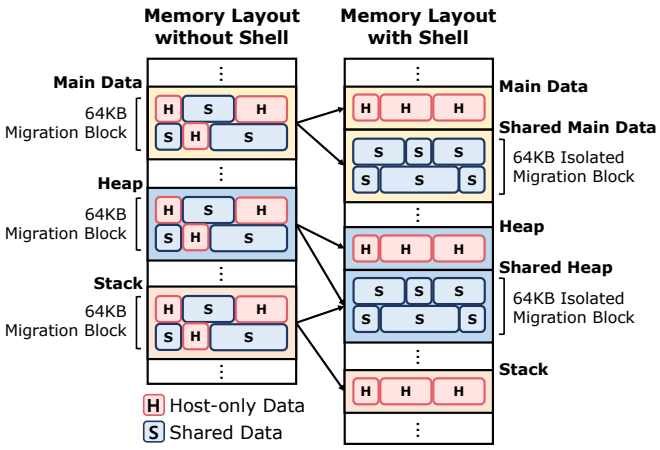


Figure 8: Memory snapshots illustrating how SHELL allocates `shared data` in isolated migration blocks to enforce fine-grained access control. The left shows the memory layout before applying SHELL, and the right shows that after applying SHELL.

and isolated from the `Main Data` region. Then SHELL places all global `shared data` variables in this isolated `Shared Main Data` region, which is instructed in the static instrumentation phase. (ii) To secure heap variables, SHELL maintains additional allocation pool (i.e., `Shared Heap`) dedicated to `shared data`, where the underlying memory page for the allocation pool is aligned as 64KB. Then all the memory allocation invocations for `shared data` are instrumented to be allocated from this dedicated pool, and the `host-only data` is allocated from the default pool. (iii) To secure stack variables, SHELL transforms all stack-based allocations into heap-based allocations within a dedicated `shared data` memory pool. Specifically, for each stack `shared data` variable, SHELL replaces its declaration with an equivalent malloc invocation in the function's prologue, and inserts a corresponding free invocation in the function's epilogue. By enforcing three types of memory allocation to be identified between `shared data` and `host-only data` and placing them in isolated pages, SHELL prevents sub-page vulnerabilities.

5. Implementation

We implemented SHELL's pre-processor in LLVM/Clang [6], and the access controller in the NVIDIA GPU driver. This section describes how we implemented SHELL in the Clang and NVIDIA GPU driver.

5.1. Implementation of Pre-processor

Customized mmap. We also customized `mmap` to allocate `shared data` that is allowed to be accessed by both the host and GPU kernel. SHELL provides the `MAP_HMM` flag to specify whether the memory is mapped as `shared data` or not. We overrode the `mmap` function to our customized `mmap`, which receives a 64-bit flag. If the `MAP_HMM` flag is set,

```
+ void* custom_mmap( ..., size_t size, uint64_t flag, ... ) {
+     if (flag & MAP_HMM) {
+         size = pad_to_64KB(size);
+         void *addr = real_mmap(..., (int)flag, ...);
+         params = {.base = addr,
+                   .size = size,};
+         ioctl(fd, SET_HMM_TABLE, NULL);
+     } else {
+         return real_mmap(..., (int)flag, ...);
+     }
+ }
```

Figure 9: Customized memory mapping with SHELL.

```
- int global_array1[SIZE1];
- int global_array2[SIZE2];

+ char *alloc_address = nullptr;
+ int  *global_array1 = nullptr;
+ int  *global_array2 = nullptr;

  int main() {
    // Allocate shared data region for global variables
+   alloc_address = (char *)mmap(0, MAXSIZE, ..., MAP_HMM, ...);
+   global_array1 = (int *)mmap(alloc_address, SIZE1, ...);
+   alloc_address += SIZE1;
+   global_array2 = (int *)mmap(alloc_address, SIZE2, ...);
+   alloc_address += SIZE2;
  }
```

(a) Instrumentation for global variable.

```
  int heap_malloc() {
-     int *array = (int *)malloc(SIZE);
+     int *array = (int *)mallocx(SIZE, NEW_ARENA);
  }
```

(b) Instrumentation for malloc.

```
  int stack_allocation() {
-     int array[SIZE];
+     int *array = (int *)mallocx(SIZE, NEW_ARENA);
  }

 int function_epilogue() {
+     free(array);
 }
```

(c) Instrumentation for stack allocation.

```
  int memory_mapping() {
-     int flag = MAP_PRIVATE | MAP_ANONYMOUS;
-     int *array = (int *) mmap( ..., flag, ... );
+     uint64_t flag = MAP_PRIVATE | MAP_ANONYMOUS | MAP_HMM;
+     int *array = (int *) custom_mmap( ..., flag, ... );
  }
```

(d) Instrumentation for memory mapping.

Figure 10: Instrumentation for host code with SHELL.

the customized `mmap` allocates memory in 64KB granularity, with 64KB alignment, and the base address and size of the allocated memory are passed to the GPU driver through ioctl to update the SHELL table.

Host Code Instrumentation. SHELL identifies the `shared data` that is accessed by both the host and GPU kernel by examining the arguments passed to the GPU kernel with backward data flow analysis of use-def chains. Figure 10 shows the instrumentation for the host code to track the pointers passed to the GPU kernel. SHELL instruments all memory allocation instructions that are possibly used by the GPU kernel to update whitelists of `shared data` in the access controller. For more details, the pass instruments the following things in the host code:

- **Global Variables:** As shown in Figure 10a, all global variables that are used as arguments to the GPU kernel are instrumented to be allocated in a separated memory region mapped with `MAP_HMM` flag. First, the memory region is allocated with `mmap` with `MAP_HMM` flag for the `shared data` in global variable (line 10). Then, all global variables are instrumented to be mapped inside the memory region allocated by `mmap`, storing each global variables without internal fragmentation (line 11-14).
- **Heap Allocation (malloc):** All possible `malloc` that are used to allocate `shared data` are instrumented to use `mallocx` instead of `malloc` as shown in Figure 10b. `mallocx` is a heap allocation API from jemalloc, which allocates memory in a separated memory arena. The region of separated memory arena used by `mallocx` is regarded as `shared data`, and the base address and size of the jemalloc memory arena is passed to the GPU driver in the initialization phase of the jemalloc memory arena.
- **Stack Memory Allocation:** As shown in Figure 10c, the stack memory allocation is instrumented to the heap allocation with `mallocx`, which allocates memory in a separated memory arena for `shared data`. At the epilogue of the function, which destroys the stack frame, `dallocx` is called to free the memory allocated by `mallocx` which is instrumented from the stack allocation.
- **Memory Mapping:** As shown in Figure 10d, the memory mapping is instrumented to use the customized `mmap` with `MAP_HMM` flag if the memory is used as `shared data`.

5.2. Implementation of Access Controller

SHELL table, which tracks the base address and size of the `shared data` that is needed by the GPU kernel. The table is kept per process, and each table's entry is updated by the instrumented host code, calling the `ioctl` to the GPU driver. SHELL table maintains the base address and size of the region of `shared data` that is used by both the host and GPU kernel. In the initial phase of the process launching a GPU kernel, the SHELL table corresponding to the process is initialized with zeroized entries, and the range of the region for the heap and data segments reserved for the `shared data` is updated to the SHELL table. While the process is running, new SHELL entries are added to the table when the host code allocates memory for `shared data` using the `mmap` function with `MAP_HMM` flag, and marked as invalid when the corresponding memory pages are unmapped. Entries for the `shared data`'s stack and data segments live for the lifetime of the process, serving as a whitelist for the `shared data` in the stack and data segments.

6. Evaluation

This section evaluates our attack's feasibility (§6.1) as well as the mitigation effectiveness of SHELL (§6.2) against the attack and its performance overhead (§6.3).

```
// libcuda_example.cu
// nvcc -Xcompiler -fPIC -shared -G -arch=sm_80

__device__
void device_code(float *data)
{
    // normal device code
}

// CUDA kernel code
__global__
void device_kernel(int len, char *h_req_id, float *data)
{
    char d_req_id[16] = {0};

    // function pointer to be overwritten
    void (*operation)(float*) = device_code;

    // if len > 16, it overwrites the function pointer
    for(int i = 0; i < len; i++)
    {
        d_req_id[i] = h_req_id[i];
    }

    operation(data);

}
```

Figure 11: Simplified code for victim's vulnerable GPU kernel

Evaluation Setup. All attacks were carried out on Ubuntu 22.04 LTS (Linux kernel 6.8.0) with an Intel Core i7-14700K CPU and an NVIDIA RTX 4060 GPU. We used the open-source NVIDIA driver version 570, and CUDA version 12.8 to test our attacks and mitigation. In §6.1, we used Python version 3.12 as a victim process to demonstrate the attack. In order to demonstrate exploitation with attacker-controlled GPU kernel, we embedded the attacker's malicious GPU kernel into the Chrome (version 138.0.7191) WebGPU platform directly, as the current Chrome WebGPU does not support general-purpose GPU kernels.

6.1. Attack on HMM-enabled GPU Kernel

Exploit vulnerable GPU kernel. We embedded a vulnerable GPU kernel function in PyTorch (which is similar to the known PyTorch vulnerability [19]), and the victim runs this vulnerable GPU kernel by calling PyTorch's API to perform basic matrix operations on the GPU. If the victim uses PyTorch's API to perform matrix operations on the NVIDIA GPU, PyTorch loads the vulnerable GPU kernel and CUDA runtime library (`libcuda-rt`) to run the GPU kernel. As shown in Figure 5, the attacker can use this vulnerability to leak the address space layout of the host process, and overwrite the control-flow relevant data, such as the return address or the `libcuda-rt`'s GOT entries. Figure 11 illustrates a simplified code of the vulnerable GPU kernel, which is a simplified version of the vulnerable GPU kernel in the PyTorch framework. The vulnerable GPU kernel copies `len` bytes from a host pointer (`h_req_id`) into a fixed-size buffer (`d_req_id`), enabling the attacker to overwrite the function pointer (`operation`). The GPU device code serves basic matrix operations, which are commonly offloaded to GPU in machine learning applications. Redirecting the pointer (`operation`) to an attacker-controlled payload causes the GPU kernel to execute the attacker's code.

We began by generating a payload: we compiled the attacker's GPU device code into a GPU executable (i.e., SASS) and embedded it into the input data (i.e., typically an image or text used in machine learning applications). We then triggered the vulnerability by passing a large `len` value to the vulnerable GPU kernel, which causes the GPU kernel to copy more bytes than the size of the buffer (`d_req_id`), and overwrite the `operation` pointer with the address of the attacker's payload. It successfully diverts the control flow of the victim GPU kernel to the attacker's payload, executing the attacker's code on the GPU. The attacker's payload is designed to overwrite the GOT entry of the `free` function in the victim process to the address of a gadget that calls the `system`. As the address space layout of the victim process is leaked by the GPU kernel, the attacker can manipulate the argument passed to the `system` function to execute arbitrary commands in the victim's system. We successfully executed this attack, demonstrating that HMM-enabled GPU kernels can indeed corrupt host memory and hijack control flow.

Exploit with attacker-controlled GPU kernel. We tested our attack on the Chrome WebGPU platform, which allows attacker-controlled GPU code to be executed on the GPU. However, as the current WebGPU implementation does not support general-purpose GPU kernels, we embedded the attacker's GPU kernel into the Chrome WebGPU platform directly to test our attack on Chrome's highly sandboxed WebGPU environment. We first programmed a GPU kernel that leaks the address space layout of the host Chrome GPU process, and then find the address of the `libcuda-rt`'s GOT entry of the `free` function and overwrites it with the address of a gadget that calls the `ioctl` function. We then embedded this GPU kernel into one of the existing rendering functions in the Chrome WebGPU platform, which is executed when the victim Chrome visits a specific website that uses WebGPU. We waited for the victim Chrome to visit the specific website and execute the attacker's GPU kernel, which successfully leaked the address space layout of the host Chrome GPU process, including the address of the `libcuda-rt`'s GOT entry of the `free` function. As Chrome does not blacklist the `ioctl` function in its GPU process sandbox, we executed arbitrary `ioctl` commands with overwriting the `libcuda-rt`'s GOT entry. It can further lead to triggering vulnerabilities in the GPU driver, which can result in serious consequences, such as privilege escalation.

6.2. Preventing GHOST-ATTACK with SHELL

Preventing exploitation of vulnerable GPU kernel. We compiled the vulnerable GPU kernels with SHELL's instrumentation and executed the exploit code that triggers the vulnerability in the victim's GPU kernel, and this GPU kernel is executed by the victim's PyTorch framework. Although SHELL cannot prevent the vulnerability in the GPU kernel from being exploited and the attacker's payload from being executed on the GPU kernel, it successfully prevents the attacker's payload from accessing the `host-only data`. It blocks attacker's attempts to leak the address space layout of

TABLE 1: Latency (in seconds) for running the ERA5 processor application, comparing application using jemalloc (`mallocx`) versus SHELL-instrumented allocator. The last column shows total time overhead relative to the baseline (%).

Scheme	Elapsed Time (s)	Relative to Baseline (%)
baseline	3.1088	100.0 %
SHELL	3.1398	100.9 %

the victim process, as the attacker's payload cannot access the `host-only data` that contains the address mapping information of the victim process. Even if the attacker knows the address of the `libcuda-rt`'s GOT entry, it cannot overwrite any `host-only data` in the victim process, as SHELL strictly separates the `shared data` from the `host-only data`, and restricts the GPU kernel from accessing the `host-only data`.

Preventing exploitation of attacker-controlled GPU kernel. As Chrome's GPU kernel launching is not yet supported, how to integrate SHELL into the Chrome WebGPU platform is still an open question. We assumed that the future WebGPU platform launches the GPU kernel with a dedicated GPU kernel launching module, which is similar to the current WebGPU implementation for supporting GPU shaders. We applied our SHELL's instrumentation to compile our dedicated GPU kernel launching module, and tested the attack on the Chrome WebGPU platform. It blocks all attempts to access the `host-only data` from the attacker-controlled GPU kernel, as the SHELL's access control mechanism prevents the GPU kernel from accessing only `shared data` regions.

6.3. Performance Evaluation on SHELL

We evaluate the performance overhead of SHELL in two aspects. First, to understand its end-to-end performance impacts, we evaluate SHELL using a real-world application that uses HMM feature. Second, we break down the performance overhead by each component of SHELL, including the instrumented allocator and the runtime access control mechanism. We repeated each experiment 100 times and reported the average latency.

End-to-End Performance Overhead. As HMM is a relatively recent feature and has not yet seen widespread adoption in production-scale applications, we selected a publicly available HMM-enabled CUDA application [4], developed by NVIDIA, for processing the ERA5 climate dataset [3]. The ERA5 processor consists of multiple memory allocation patterns, including `mmap()` and `malloc()` for `shared data` allocation. We instrumented `mmap` call for `shared data` to use our custom flag, `MAP_HMM`, which allows the GPU driver to keep track of the memory regions of `shared data`. Since our customized allocator is built on top of the `jemalloc` library, we compared the performance of our SHELL-enabled application against a version using the `jemalloc` library. Table 1 shows the total elapsed time of process measured by `perf`. The SHELL-enabled application shows only a slight overhead of 0.9% compared to the baseline application. The memory allocation overhead is measured by the `nvidia-smi` tool [8], which can show the allocated memory size in a GPU memory for a given process, and both SHELL-enabled and baseline application show the same memory allocation size, which is 5,824 MiB. As pages migrate between the host and the GPU in 64KB units, there's no extra fragmentation in the SHELL-enabled application.

TABLE 2: Total number of each type of memory allocation on `shared data` region while running the ERA5 processor application.

Scheme	Total Number of Invocations (times)	Pecentages (%)
mmap	3	0.2 %
malloc	1,440	99.8 %

TABLE 3: Latency (in micro-seconds) for `mmap` allocations, comparing the raw baseline allocator versus SHELL-instrumented. The last column shows overhead relative to the baseline (%).

Scheme	Elapsed Time (us)	Relative to Baseline (%)
mmap	5.512	100.0 %
mmap + ioctl	6.106	110.7 %

Micro Performance Overhead. We first profiled the pattern of memory allocation in the ERA5 processor application, and found that all `shared data` allocations are done using the `mmap()` and `malloc()` functions. Table 2 shows the total number of each type of memory allocation on the `shared data` region while running the ERA5 processor application. The `mmap()` is used only for a small number of allocations, while the `malloc()` is used for the majority of allocations, consisting of 99.8% of the total allocations to the `shared data` region.

As SHELL's performance overhead mainly comes from the (1) custom `mmap()`, which updates the SHELL table with ioctl when allocating a `shared data` region, and (2) the page fault handler, which checks the access control while the process is running. Table 3 shows the latency of classical `mmap()` and SHELL's `mmap()` with `ioctl()` call. The classial `mmap()` 5.512 us, while the SHELL's `mmap()` takes 6.104 us, which shows 10.7% overhead in the SHELL's `mmap()` call. However, this overhead is negligible in the context of the ERA5 processor application, as the `mmap()` is used only for a small number of allocations (3 times in total). Table 4 shows the latency of the page fault handler in the GPU driver, comparing the raw baseline handler versus the access control enabled SHELL's page fault handler. The baseline page fault handler takes 97.02 us, while the SHELL's page fault handler takes 108.75 us, which shows 12.1% overhead in the SHELL's page fault handler.

7. Discussion

This section discusses several mitigation alternatives and future research directions.

TABLE 4: Latency (in micro-seconds) for handling page fault in the GPU driver, comparing the raw baseline handler versus the access control enabled SHELL's page fault handler. The last column shows overhead relative to the baseline (%).

Scheme	Elapsed Time (us)	Relative to Baseline (%)
baseline	97.02	100.0 %
SHELL	108.75	112.1 %

7.1. Mitigation Alternatives

Unified Virtual Memory (UVM). One straightforward way to prevent the GHOST-ATTACK is to disable HMM and instead use Unified Virtual Memory (UVM) [36]. UVM is secure but requires programmers to manually annotate data shared between the host and GPU (e.g., through `cudaMallocManaged()` API). In contrast, HMM with our proposed mitigation, SHELL, significantly reduces programmer burden while still providing security against the GHOST-ATTACK.

Fork and Sandbox. Another approach is to fork a new process and apply least-privilege (e.g., sandboxing) constraints to the child process executing the untrusted GPU kernel. While this offers security barriers at some extent, this does not fundamentally prevent the GHOST-ATTACK. The child process remains vulnerable to arbitrary host code execution caused by untrusted GPU kernels. Any channel that allows untrusted GPU kernels to trigger host execution continues to pose a security risk, even in a sandboxed environment as discussed in §3.3. In contrast, SHELL fundamentally prevents GHOST-ATTACK by blocking GPU access to sensitive host memory regions, thereby frustrating any attempts by untrusted GPU kernels to trigger host code execution.

Randomize Address Mapping for nvidiactl. A possible mitigation is to randomize the address mapping for the `nvidiactl` device file. This makes it harder for an attacker to infer the host heap address, complicating attempts to exploit the mapping of host memory. Nevertheless, this is not secure in practice. An attacker can still leak host address information through other channels, such as host pointers passed as the GPU kernel arguments in the HMM model. Thus, this approach is not a fundamental solution to the GHOST-ATTACK. In contrast, SHELL allows the GPU to access only the data shared between the GPU and CPU through our mitigation, while preventing access to host-only data. As a result, SHELL successfully blocks GHOST-ATTACK from leaking additional host address layout information, which resides in host-only memory regions accessible only by the host.

7.2. Future Research Directions

Broaden Secure HMM Support to Other Platforms. Current GHOST-ATTACK and its mitigation, SHELL, are focused on NVIDIA's CUDA platform. However, the design principles of device drivers to support HMM vary across hardware and its vendors. For example, the Linux kernel provides HMM helper functions to support both memory mirroring and memory migration. NVIDIA GPU driver adopts a memory migration model, however, in other platforms, memory mirroring, which holds a copy of the host memory in the GPU memory instead of migrating it, can be used. Furthermore, the AMD, which is another major GPU vendor, decided to support HMM in its general-purpose GPU programming platform, ROCm, since its version 6.5.0, which is not yet released at the time of this writing. To broaden the applicability of SHELL, examining the design of HMM support in other platforms can be a future work.

GPU-to-Host Attack with Shader. The GHOST-ATTACK in this paper targets the general-purpose GPU programming model. However, the shader programming model, which is widely used in graphics applications, can also be a good vector for GPU-to-Host memory attacks. For example, in integrated GPU systems, as the GPU shares the same physical memory with the host, several graphics backends, such as Vulkan [14] and DirectX [2], provide APIs to declare `shared data` that both the host and GPU can access directly. This `shared data` region is supported in a more constrained manner than HMM, but it is worth investigating whether the GPU-to-Host memory attack can be extended to the shader programming model.

8. Related work

Prior research on GPU security has predominantly focused on attacks confined to GPU memory, which has traditionally been isolated from host memory in discrete GPU architectures. A broad class of attacks has been demonstrated, including side-channel leakage, memory corruption, and privilege escalation within the GPU memory domain [31, 25, 46, 47, 19, 20, 40, 48, 43, 45, 29]. To defend against these threats, a number of mitigation strategies have been proposed to detect or prevent GPU memory vulnerabilities [24, 44, 21].

GPU Memory Disclosure. Numerous studies have shown that GPUs are vulnerable to various forms of memory disclosure through both microarchitectural side channels and direct memory access vulnerabilities. Naghibijouybari et al. [31] demonstrated that shared GPU resources can be exploited to build covert channels between co-resident GPU kernels, enabling information leakage. Lee et al. [25] uncovered that GPU memory is not reliably cleared after deallocation. They showed that by allocating large GPU buffers, an attacker can retrieve residual graphical data, such as textures rendered by Chromium, and reconstruct visual information about previously visited webpages. Zhang et al. [46] reverse-engineered the TLB behavior of NVIDIA GPUs to break the isolation guarantees of multi-instance GPUs (MIG) and infer cross-partition memory access patterns. Zhang et al. [47] introduced a timer-free side-channel using GPU cache management instructions and leaked the data without relying on high-precision timers. Karimi et al. [23] demonstrated that timing side-channels in mobile GPUs can be exploited to leak sensitive information, such as AES-128 keys.

GPU Memory Corruption. Several studies have demonstrated that memory-safety bugs in GPU kernels can be exploited to corrupt GPU memory, leading to integrity violations, model degradation, and even control flow hijacking on the GPU. Di et al. [19] demonstrated that classic memory-safety bugs in GPU kernels can be exploited to corrupt GPU memory and compromise the integrity of GPU applications. Park et al. [40] demonstrated that malicious manipulation of the GPU memory can degrade deep-learning model outputs to random guessing, revealing a new threat to ML integrity. Guo et al. [20] showed the GPU memory corruption attacks on NVIDIA GPUs, which exploit memory safety bugs in GPU kernels to corrupt GPU memory. Through extensive reverse-engineering, they disclosed that the return addresses of GPU kernels can be overwritten to redirect the execution flow of the GPU kernel. They further noted that the modern GPU lacks common security features such as Write XOR Execute (W^X) and stack canaries. Finally, they demonstrated that classic exploit techniques in host memory, including buffer overflow and return-oriented programming (ROP), can be adapted to hijack the control flow of GPU kernels as well.

Limitations of Previous GPU Attacks. Although these studies have demonstrated a wide range of side-channels and memory corruption attacks on GPU memory, they share a common limitation: none can directly access or corrupt host memory. In every case, the attacker's capabilities are confined to GPU memory, which, under traditional GPU architectures, is isolated from the host program's memory. Consequently, even the most powerful GPU-side exploit cannot read or overwrite sensitive data stored in the host program's memory.

In contrast, our work breaks this long-standing limitation with HMM. We introduce GHOST-ATTACK, a new class of attacks that exploit memory-safety bugs in HMM-enabled GPU kernels to directly read from and corrupt the host program's memory. These GPU-to-host memory corruption attacks violate a core security assumption of the CUDA programming model, namely, that GPU execution cannot affect host memory integrity. By breaching this boundary, GHOST-ATTACK enables a new threat model in which attackers can break host ASLR, exfiltrate sensitive data, and even hijack the host program's control flow from within GPU code.

Mitigations for GPU Memory Safety. To address memory corruption vulnerabilities in GPU kernels, several compiler- and runtime-based techniques have been proposed. Henriksen et al. [21] proposed a compiler-based approach that inserts bounds checks on array accesses in GPU kernels to mitigate out-of-bounds memory accesses. Tarek et al. [44] developed CUCATCH, a dynamic analysis tool designed to detect both spatial and temporal memory errors in GPU kernels. Their approach efficiently instruments kernel code to catch a wide range of memory safety violations, offering developers practical support for debugging and validation. Lee et al. [24] proposed a region-based memory safety analysis framework that performs static pointer analysis at compile time and inserts runtime bounds checks to prevent spatial memory violations.

While effective within the GPU memory domain, these techniques assume strict memory isolation between the GPU and the CPU. As such, they are not designed to protect against the broader class of GPU-to-host memory attacks enabled by HMM. In contrast, our SHELL approach addresses this emerging threat by enforcing access control across the unified memory space.

9. Conclusion

In this paper, we expose a critical blind spot in NVIDIA's Heterogeneous Memory Management: GPU kernels have been granted implicit, unlimited access to host memory under the assumption that they can be fully trusted. We show that this trust is fundamentally misplaced by demonstrating existing memory-safety bugs and attacker-supplied GPU kernels can lead to compromise of host memory, allowing arbitrary read and write on host memory. To address this, we introduce SHELL, a lightweight Clang/LLVM-based instrumentation and driver-enforced access-control framework that restricts GPU memory accesses strictly to compiler identified `shared data` regions. Our prototype integration with an open-source NVIDIA GPU driver shows that SHELL completely blocks GHOST-ATTACK with negligible performance overhead. As HMM gains traction in modern GPU programming, SHELL's compile-time identification of `shared data` and runtime enforcement model provide a practical blueprint for securing future memory management in heterogeneous computing systems.

10. Acknowledgment

We are grateful to the anonymous reviewers for their valuable feedback and constructive suggestions, which have significantly improved the final version of this paper. This work was supported by the National Research Foundation (NRF) of Korea grant funded by the Korean government (MSIT) (No. RS-2023-00209093). This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korean government (MSIP) (No.2020-0-01840, Analysis on technique of accessing and acquiring user data in smartphone). This work was supported by Samsung Electronics Co., Ltd (IO221213-04119-01). The Institute of Engineering Research (IOER) and Automation and Systems Research Institute (ASRI) at Seoul National University provided research facilities for this work.

References

[1] Chromium graphics // chrome gpu. https://www.chromium.org/developers/design-documents/chromium-graphics/.
[2] Direct3d - microsoft. https://learn.microsoft.com/ko-kr/windows/win32/direct3d.
[3] Ecmwf reanalysis v5 (era5), . https://www.ecmwf.int/en/forecasts/dataset/ecmwf-reanalysis-v5.
[4] Cuda 12.2 hmm demos, . https://github.com/NVIDIA/HMM_sample_code.
[5] jemalloc memory allocator. https://jemalloc.net/.
[6] Llvm clang. https://clang.llvm.org/.

[7] Nvidia cuda compiler driver nvcc. https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/.

[8] System management interface smi. https://developer.nvidia.com/system-management-interface.

[9] Opencl overview - the khronos group inc, . https://www.khronos.org/opencl/.

[10] Opengl - the industry standard for high performance graphics, . https://www.opengl.org/.

[11] Parallel thread execution isa version 8.8. https://docs.nvidia.com/cuda/parallel-thread-execution/.

[12] Pytorch. https://pytorch.org/.

[13] Tensorflow. https://www.tensorflow.org/.

[14] Vulkan-cross platform 3d graphics. https://www.vulkan.org/.

[15] Web neural network api. https://www.w3.org/TR/webnn/.

[16] M. Alex, S. Vargaftik, G. Kupfer, B. Pismeny, N. Amit, A. Morrison, and D. Tsafrir. Characterizing, exploiting, and detecting dma code injection vulnerabilities in the presence of an iommu. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 395–409, 2021.

[17] AMD. Unified memory management. https://rocm.docs.amd.com/projects/HIP/en/docs-develop/how-to/hip_runtime_api/memory_management/unified_memory.html.

[18] compsec snu. ghost-in-the-shell, 2025. https://github.com/compsec-snu/ghost-in-the-shell.

[19] B. Di, J. Sun, and H. Chen. A study of overflow vulnerabilities on gpus. *2023 22nd RoEduNet Conference: Networking in Education and Research (RoEduNet)*, pages 103–115, 2016. doi: 10.1007/978-3-319-47099-3_9. URL https://doi.org/10.1007/978-3-319-47099-3_9.

[20] Y. Guo, Z. Zhang, and J. Yang. Gpu memory exploitation for fun and profit. In *Proceedings of the 33rd USENIX Security Symposium (Security)*, Aug. 2024.

[21] T. Henriksen. Bounds checking on gpu. *International Journal of Parallel Programming*, 49(6):761–775, 2021. doi: 10.1007/s10766-021-00703-4.

[22] jwnhy. Illegal memory access with adaptiveavgpool2d, 2025. https://github.com/pytorch/pytorch/issues/145349.

[23] E. Karimi, Z. H. Jiang, Y. Fei, and D. Kaeli. A timing side-channel attack on a mobile gpu. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*, pages 67–74, 2018.

[24] J. Lee, Y. Kim, J. Cao, E. Kim, J. Lee, and H. Kim. Securing gpu via region-based bounds checking. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 27–41, 2022.

[25] S. Lee, Y. Kim, J. Kim, and J. Kim. Stealing webpages rendered on your browser by exploiting gpu vulnerabilities. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2014.

[26] R. Li. cublas address out of bounds for particular matrix size, 2020. https://stackoverflow.com/questions/61979677/cublas-address-out-of-bounds-for-particular-matrix-size.

[27] lights0123. hipscript, 2025. https://github.com/lights0123/hipscript/.

[28] Linux. Heterogeneous memory management(hmm). https://www.kernel.org/doc/html/v5.0/vm/hmm.html.

[29] H. T. Maia, C. Xiao, D. Li, E. Grinspun, and C. Zheng. Can one hear the shape of a neural network?: Snooping the gpu via magnetic side channel. In *Proceedings of the 31st USENIX Security Symposium (Security)*, Aug. 2022.

[30] S. Mittal, S. Abhinaya, M. Reddy, and I. Ali. A survey of techniques for improving security of gpus. *Journal of Hardware and Systems Security*, 2:266–285, 2018. doi: 10.1007/s41635-018-0039-0.

[31] H. Naghibijouybari, A. Neupane, Z. Qian, and N. Abu-Ghazaleh. Rendered insecure: Gpu side channel attacks are practical. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, Canada, Oct. 2018.

[32] NVIDIA. cublas, . https://developer.nvidia.com/cublas.

[33] NVIDIA. Cuda runtime api :: Cuda toolkit documentation, . https://docs.nvidia.com/cuda/cuda-runtime-api/index.html.

[34] NVIDIA. Nvidia cudnn - cuda deep neural network, . https://developer.nvidia.com/cudnn.

[35] NVIDIA. Nvidia cufft, . https://developer.nvidia.com/cufft.

[36] NVIDIA. Unified memory for cuda beginners, . https://developer.nvidia.com/blog/unified-memory-cuda-beginners/.

[37] NVIDIA. Simplifying gpu application development with heterogeneous memory management, 2023. https://developer.nvidia.com/blog/simplifying-gpu-application-development-with-heterogeneous-memory-management/.

[38] NVIDIA. Nvidia compute sanitizer, 2025. https://docs.nvidia.com/compute-sanitizer/ComputeSanitizer/index.html#.

[39] NVIDIA. Cuda gdb, 2025. https://docs.nvidia.com/cuda/cuda-gdb/index.html.

[40] S. Park, O. Kwon, Y. Kim, S. Cha, and H. Yoon. Mind control attack: : Undermining deep learning with gpu memory exploitation. *Computers & Security*, 102:102115, Nov 2020. doi: 10.1016/j.cose.2020.102115.

[41] R. D. Pietro, F. Lombardi, and A. Villani. Cuda leaks: A detailed hack for cuda and a (partial) fix. *ACM Trans. Embed. Comput. Syst.*, 15(1):15–40, jan 2016. doi: 10.1145/2801153.

[42] shallowbeach. Webgpu fundamentals. https://webgpufundamentals.org/webgpu/lessons/webgpu-fundamentals.html.

[43] H. Taneja, J. Kim, J. J. Xu, S. Van Schaik, D. Genkin, and Y. Yarom. Hot pixels: Frequency, power, and temperature attacks on {GPUs} and arm {SoCs}. In *Proceedings of the 32nd USENIX Security Symposium (Security)*, Aug. 2023.

[44] M. Tarek Ibn Ziad, S. Damani, A. Jaleel, S. W. Keckler, and M. Stephenson. Cucatch: A debugging tool for efficiently catching memory safety violations in cuda applications. *Proceedings of the ACM on Programming Languages*, 7(PLDI):124–147, 2023. doi: 10.1145/3591225.

[45] Z. Zhan, Z. Zhang, S. Liang, F. Yao, and X. Koutsoukos. Graphics peeping unit: Exploiting em side-channel information of gpus to eavesdrop on your neighbors. In *Proceedings of the 43rd IEEE Symposium on Security and Privacy (Oakland)*, SAN FRANCISCO, CA, May 2025.

[46] Z. Zhang, T. Allen, F. Yao, X. Gao, and R. Ge. Tunnels for bootlegging: Fully reverse-engineering gpu tlbs for challenging isolation guarantees of nvidia mig. In *Proceedings of the 30th ACM Conference on Computer and Communications Security (CCS)*, Nov. 2023.

[47] Z. Zhang, K. Cai, Y. Guo, F. Yao, and X. Gao. Invalidate+compare: A timer-free gpu cache attack primitive. In *Proceedings of the 33rd USENIX Security Symposium (Security)*, Aug. 2024.

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

Appendix A.
Meta-Review

The following meta-review was prepared by the program committee for the 2026 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

A.1. Summary

This paper exposes a new attack surface introduced by Linux Heterogeneous Memory Management (HMM), where host memory is mapped into the GPU address space even when not needed for benign computation. The authors demonstrate GHOST-ATTACK, a GPU-originated exploitation technique that escalates vulnerabilities in CUDA GPU kernels to hijack host program execution, bypassing ASLR and compromising applications like PyTorch. The root cause lies in memory bugs that enable malicious GPU kernels to access arbitrary host memory. To mitigate this, the paper proposes SHELL, a compiler-based defense built on LLVM, which detects shared memory regions and enforces protection through GPU driver page-fault handling. Evaluation shows that SHELL effectively defends against GHOST-ATTACK with low performance overhead.

A.2. Scientific Contributions

- Identifies an impactful vulnerability
- Independent Confirmation of Important Results with Limited Prior Research

A.3. Reasons for Acceptance

1) The paper presents the first ever attack against the host's memory from a GPU kernel, and demonstrates a critical vulnerability in HMM.
2) Open sourcing this work would significant benefit the community.
3) Authors provide a practical defense against the newly discovered attack, using the popular LLVM framework.