Demystifying and Exploiting ASLR on NVIDIA GPUs

Ruofan Zhu*, Ganhao Chen*, Wenbo Shen*†, Lyuye Zhang‡, Dakun Shen*, Rui Chang*, Yanan Guo§
*Zhejiang University, Hangzhou, China
‡Nanyang Technological University, Singapore
§University of Rochester, Rochester, USA
{zhuruofan, chenganhao, shenwenbo, dakun, crix1021}@zju.edu.cn, zh0004ye@e.ntu.edu.sg, yguo51@cs.rochester.edu

***Abstract*—GPUs are foundational to modern AI workloads, powering deep learning training and inference. As their deployment becomes increasingly widespread, GPUs have also emerged as attractive targets for attackers. To strengthen their defenses, security measures, such as Address Space Layout Randomization (ASLR), are deployed. However, in contrast to the extensive research on CPU ASLR, in-depth studies of GPU ASLR are still missing.**

This paper presents the first comprehensive examination of ASLR on NVIDIA GPUs. We propose two novel techniques to thoroughly inspect memory mappings and collect randomized GPU addresses at scale. Leveraging these techniques, we construct a fine-grained GPU memory map and introduce entropy-based metrics to quantify the strength of randomization. Our study uncovers multiple previously unknown weaknesses of ASLR on NVIDIA GPUs, including an unrandomized GPU heap and correlated ASLR offsets between GPU and CPU regions, which undermine the security of both GPU and CPU ASLR. These findings have been confirmed by NVIDIA. Furthermore, we conduct a practical case study demonstrating how these weaknesses can be exploited to infer CPU ASLR offsets from the GPU. Finally, we give mitigations to enhance GPU ASLR security.

1. Introduction

Graphics Processing Units (GPUs) play a critical role in artificial intelligence by accelerating the computation required for training and running machine learning models. Their massively parallel architecture consists of thousands of smaller, more efficient cores that deliver high throughput. This design is essential for training complex neural networks. Among all GPUs, NVIDIA GPUs have consistently maintained a dominant position in the discrete GPU market, often holding over 90% of the market share [11] for their strong performance and extensive software ecosystem.

As NVIDIA GPUs become increasingly prevalent, their security becomes more critical. To enhance security, NVIDIA GPUs have adopted mechanisms originally developed for CPUs. One measure NVIDIA has implemented is Address Space Layout Randomization (ASLR) [46], which has been applied to its GPUs to enhance security. ASLR mitigates certain attacks by randomizing memory addresses used by critical components, which makes it more difficult for attackers to locate sensitive memory regions.

While CPU ASLR has been extensively studied [5], [10], the implementation details and security implications of GPU ASLR remain largely unexplored. Earlier research by Zhang et al. [48] and Mittal et al. [29] concluded that ASLR was not implemented on GPUs. Subsequently, Peng et al. [38] were the first to implement ASLR in GPU allocators, marking a significant development in GPU memory security. Hoover et al. [18] and Cismaru et al. [6] experimentally confirmed the presence of ASLR on NVIDIA GPUs, but they did not conduct an in-depth analysis. Park et al. [37] further discovered that GPU ASLR randomizes code and data segments while leaving the CUDA GPU runtime libraries without randomization. However, their work primarily focused on GPU memory manipulation rather than a comprehensive analysis of GPU ASLR. Consequently, a complete examination of ASLR on NVIDIA GPUs is still lacking.

To address this research gap, this paper conducts the first comprehensive study of ASLR on NVIDIA GPUs. Specifically, we first reconstruct the GPU memory layout and then reverse-engineer the semantics of different memory regions to obtain a comprehensive view of the GPU memory layout. Next, we define two ASLR randomness metrics and conduct experiments to collect and measure both the entropy of GPU memory layout randomization and its correlation with the corresponding CPU memory layout.

To achieve these, we must address several technical challenges. The first challenge is understanding the previously unknown semantics of various memory blocks, which is complicated by the black-box nature of NVIDIA GPUs. Several memory regions are hidden from the user and cannot be accessed or observed through conventional debugging tools such as `cuda-gdb`. As a result, researchers must rely on reverse engineering to uncover the semantics of these regions. The second challenge is efficiently collecting large volumes of randomized addresses for ASLR analysis. Techniques commonly used on CPUs, such as printing addresses with `printf` or `printk` [5], are not applicable on GPUs because

†*Wenbo Shen is the corresponding author.*

critical regions such as `CUDA library` and `.text` segments cannot be directly accessed or observed. An alternative is to extract randomized addresses from GPU page tables during every execution. However, this requires dumping the entire GPU physical memory and scanning it page by page to locate and decode page tables, which is prohibitively inefficient and impractical at scale.

To address these challenges, we propose two novel techniques: *FlagProbe* for in-depth analysis of GPU memory layouts, and *AnchorTrace* for efficient collection of ASLR addresses. These techniques enable a comprehensive analysis of GPU memory layout and ASLR, revealing previously unknown weaknesses in NVIDIA's ASLR implementation.

Our findings. For GPU memory layout, all CUDA processes share two large GPU memory regions and also map a CPU kernel-space page into the user-privileged GPU processes' memory layout. In addition, both the CPU and GPU map some physical memory pages into their respective virtual address spaces, but with different permissions: the GPU has RWX access, while the CPU is restricted to RW. *These shared memory regions among CUDA processes facilitate covert channel attacks, and the GPU-CPU memory sharing with differing permissions enables potential cross-processor attacks—such as injecting GPU-executable code from the CPU or crafting malicious data on the GPU to compromise the CPU.*

We also observe that even with GPU ASLR enabled, the GPU heap remains entirely unrandomized. Most other GPU memory regions (including `.text` and `CUDA library`) are randomized with the same ASLR offset. Moreover, certain CPU memory regions (e.g., `glibc`) exhibit a correlated ASLR offset with GPU regions, and their difference only has 7 bits of entropy. *This unrandomized heap and coarse-grained randomization undermine GPU ASLR's effectiveness; the ASLR offset correlation further allows attackers to infer the CPU ASLR offset from the GPU.*

Building on these findings and the identified weaknesses of GPU ASLR, we present a practical case study that exploits a simple stack out-of-bounds (OOB) read to leak the GPU ASLR addresses and then leverages the correlated offsets to infer the CPU ASLR offset. This demonstrates that GPU ASLR is not only limited in its own security, but also undermines the ASLR protections of the CPU. Therefore, this work, with its novel techniques, new findings, and practical exploitation, further advances the state-of-the-art of GPU ASLR. In summary, we make the following contributions:

- **New study**. We conduct the first systematic analysis of the ASLR on NVIDIA GPUs, including the comprehensive memory layout with semantics and ASLR implementation.
- **Novel techniques and framework**. We propose novel techniques for analyzing GPU memory layout and ASLR, and have developed and open-sourced the associated analysis framework at https://github.com/ZJU-SEC/NvidiaASLR.
- **New findings and weaknesses**. We present seven new findings that expose previously unknown weaknesses in NVIDIA GPU memory layout and ASLR. These findings have been confirmed by NVIDIA.

Ethics Considerations. We responsibly disclosed identified ASLR weaknesses to NVIDIA, which acknowledged and confirmed the issues in January 2025. Notably, one of our findings—GPU address leakage from the hidden stack frame (Finding-7)—has been fixed in the recently released driver version 570.

To further reduce potential risks for users of affected systems, we have anonymized all sensitive address values presented in the paper to prevent direct misuse by malicious actors. Additionally, our released artifact intentionally omits key exploit details. Finally, we provide practical and effective mitigation strategies to help users secure impacted systems.

2. Background and Motivation

In this section, we present background knowledge related to NVIDIA GPUs and our motivation. We first describe the GPU programming model in §2.1. We then introduce how GPUs access memory §2.2. Finally, we illustrate our motivations §2.3. We use NVIDIA-specific terminology for descriptions.

2.1. NVIDIA GPU Programming Model

Modern GPUs adopt a parallel execution model tailored for high-throughput, data-parallel tasks. NVIDIA GPUs, in particular, follow the Compute Unified Device Architecture (CUDA) programming model, which exposes the massive parallelism of GPU hardware to developers through a C-like interface.

Under this model, the CPU (host) and GPU (device) maintain separate memory spaces and manage different page tables. Computation is offloaded to the GPU by launching kernels, which are special functions executed by a large number of lightweight threads in parallel. Threads are organized hierarchically into warps (typically 32 threads), thread blocks, and grids. Each thread executes the same kernel code but operates on different data.

Importantly, from a systems and security perspective, the GPU's memory management, context switching, and kernel execution are orchestrated by a combination of software (CUDA runtime and driver) and hardware like GPU System Processor (GSP), which is a RISC-V processor on newer NVIDIA GPUs. These layers are largely closed-source and operate in a separate address space from the CPU, posing challenges for memory introspection and security analysis.

2.2. NVIDIA GPU Memory Access Model

NVIDIA GPUs are equipped with their own Graphics Memory Management Unit (GMMU), which manages virtual-to-physical address translation using a GPU-specific page table format. This translation capability allows GPUs

to resolve addresses for their own memory, CPU memory, and even peer GPU memory.

GPU access GPU memory (video memory). When accessing its own physical memory, the GPU relies on its GMMU and page tables. Specifically, the GMMU performs a page table walk to resolve GPU virtual addresses into GPU physical addresses. Once the translation is complete, the GPU directly accesses the corresponding memory regions.

GPU access CPU memory (system memory). When accessing CPU memory, the GPU uses its GMMU to translate virtual addresses into I/O virtual addresses (IOVA), which are then used for DMA transactions. If the IOMMU is disabled, the IOVA corresponds directly to the CPU physical address. If enabled, the IOMMU further translates the IOVA into the actual CPU physical address. Data transfer is then performed via PCIe using the resolved address.

GPU access peer GPU memory (peer memory). In peer memory access, the GPU uses its GMMU to resolve virtual addresses to peer GPU physical addresses. The actual memory access is performed using interconnect technologies such as GPU Direct [42], NVLink [13], or NVSwitch [26], which offer low-latency, high-bandwidth communication.

2.3. Motivation

NVIDIA GPUs are widely used for high-performance computing tasks, such as deep learning and graphics rendering. With the increasing computational demands, GPU security has become more critical. Following the security practices of CPUs, NVIDIA has adopted ASLR as a key defense to reduce memory corruption risks. While CPU ASLR has been extensively studied, the implementation of GPU ASLR on NVIDIA platforms and its relationship with CPU-side ASLR remain largely unexplored.

Existing studies of ASLR on NVIDIA GPUs are limited in scope and depth. Some studies focus only on specific memory regions' layout, such as the stack [17] or CUDA libraries [37], without comprehensively analyzing the overall GPU memory layout. Others merely verify the presence of ASLR on NVIDIA GPUs [48], [29], [18], [6], but do not delve into its detailed implementation. Additionally, some research assumes that GPU ASLR has already been bypassed and focuses on side-channel [49] or memory corruption attacks [17], without analyzing the ASLR mechanisms themselves. As a result, this paper aims to fill this gap by systematically analyzing the design and effectiveness of NVIDIA GPU ASLR. In particular, we investigate the GPU memory layout, the design and implementation of ASLR, its entropy, its correlation with CPU-side ASLR, and the potential for information leakage or bypass.

To the best of our knowledge, although NVIDIA has open-sourced its GPU kernel modules [8], the details of memory management mechanisms for CUDA applications remain mainly closed-source. These mechanisms are implemented within the GPU System Processor (GSP) firmware, which runs on a RISC-V processor inside the GPU [36]. Consequently, our analysis requires **black-box reverse engineering effort**, which needs to be conducted without access

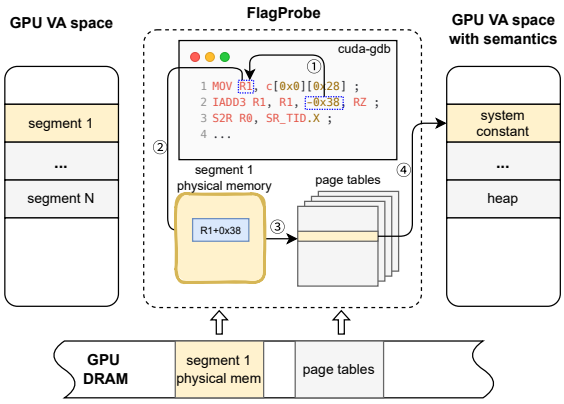


Figure 1: FlagProbe technique for identifying segments' semantics in GPU VA space.

to internal documentation or source code. We believe this paper is **the first work to comprehensively reveal the detailed structure of the GPU virtual address space on NVIDIA platforms**, including which memory segments are mapped and how they are randomized. Building on this knowledge, we further **reverse-engineer the detailed behavior of ASLR within the GPU virtual address space**. This understanding is crucial for both researchers and adversaries, as it provides a foundation for analyzing GPU memory behavior and exploring potential security vulnerabilities.

3. Techniques Design and Implementation

This section presents the core techniques and the analysis framework developed in our work. We first introduce the two key challenges we encountered during our analysis and describe a corresponding technique designed to address them in §3.1 and §3.2, respectively. We then present the overall workflow of our analysis framework in §3.3, which integrates these techniques to study GPU ASLR. Note that this section focuses on methodology; the detailed results are presented in §4 and §5, including the reconstructed GPU memory layout and a thorough analysis of the ASLR implementation.

3.1. FlagProbe: Semantic Reverse Engineering of GPU Memory via Crafted Flag Probing

To understand the implementation of ASLR on black-box NVIDIA GPUs, a necessary first step is to reconstruct the GPU's virtual memory layout before any randomization is applied. Specifically, we must identify what memory segments are mapped into the GPU virtual address space, what each segment is used for, and in what order these segments appear. However, this information has not been documented or studied prior to our work. The layout and semantics of various segments are hidden behind closed-source drivers/firmware and runtime systems. While GPU

page tables reveal segment mappings and corresponding addresses, they do not provide any semantic information about the regions (e.g., `.text`, `heap`). Therefore, as a necessary first step toward demystifying GPU ASLR, we aim to reverse-engineer the semantic layout of GPU virtual memory without randomization.

Challenge 1. Comprehensively identifying the semantics of GPU virtual memory segments is non-trivial. First, NVIDIA's GPU software stack is largely a black box. The vendor does not disclose how the runtime organizes memory or labels memory segments internally. This forces us to infer the layout using limited ground truth and without official documentation. Second, traditional reverse engineering methods have limitations as many critical regions in the CUDA address space are entirely invisible to users. For example, segments such as `system constant memory` store sensitive runtime-managed data and are inaccessible via `printf` or tools like `cuda-gdb`. These memory regions cannot be simply inspected using traditional debugging or printing techniques. Third, even for memory regions that are observable, the information retrieved via traditional print-based methods is often unreliable. For instance, printing a stack pointer in CUDA code typically yields an address from the `local stack`, which is not recorded in the GPU page tables. This `local stack` address is part of a per-thread illusion optimized for fast access, and different threads always print the same address [3]. In fact, the actual memory that stores each thread's stack resides in a separate `global stack` region [17]. Thus, accurate semantic recovery of memory segments requires low-level memory inspection beyond what print-based methods can offer.

Our technique. To address these, we propose FlagProbe, a technique that reverse-engineers the semantic meaning of GPU memory segments through carefully crafted flags and memory probing. A flag is a distinguishable constant, instruction, or binary pattern that is strategically embedded into GPU kernel code or naturally present in the memory. These flags are specifically designed to match the expected usage and visibility constraints of each memory segment. During the kernel execution, we scan the raw GPU physical memory to locate the runtime manifestation of these flags and identify their physical addresses. By cross-referencing these addresses with the GPU page tables, we can recover the corresponding virtual addresses and associate them with their functional semantics accordingly. In our design, we prepare multiple flag candidates with low searching collision probability. To further confirm correctness of a flag's location when a collision happens, we use different flags across multiple executions to cross-validate that the corresponding memory consistently changes.

As shown in Figure 1, we use the `system constant memory` region as an illustrative example. This region is a sensitive, undocumented memory segment managed internally by the CUDA runtime and cannot be directly accessed or modified by user code. However, by disassembling compiled CUDA kernel functions, we observed that each kernel's entry point consistently includes the following instruction sequence: `MOV R1, c[0x0][0x28]; IADD3 R1, R1, -0x38, RZ`. These instructions load the stack base address from a fixed offset within `system constant memory` and allocate stack size. We figure out the virtual memory region for the `system constant memory` through four steps. ① While breakpoints cannot be inserted between these instructions, we place a breakpoint at the subsequent instruction and use `cuda-gdb` to capture the runtime value of register `R1`. By reversing the calculation, we infer that the original value read from `system constant memory` was `R1+0x38`. ② We then scan GPU physical memory to locate this value, thereby recovering its physical address. ③ Using page tables extracted from GPU physical memory, we map this physical address back to its GPU virtual address (segment 1 in Figure 1). ④ Now, we know that segment 1 in the virtual memory is for the `system constant memory`. If there is a collision in the memory for the searched flag, we will select another candidate flag, such as the value extracted from the instruction `MOV R11, c[0x0][0x110]`, to ensure the uniqueness and correctness of the searched flag.

Similarly, we identify different memory segments using customized flags and reverse engineering techniques. Below are the strategies we employed for each segment:

- **`.text`:** We identify the GPU `.text` segment by locating a known instruction pattern. For example, the high 64 bits of the instruction `MOV R1, c[0x0][0x28]` is `0x00000a0000017a02`, which we use as a dedicate flag to locate the code segment in GPU virtual memory.
- **CUDA library:** Since printing addresses of CUDA internal functions (e.g., `memset`) using `printf` yields non-address values (e.g., `0x6e0`), we cannot locate these functions directly. Instead, we use the call instruction opcode `0x7943` as a flag. By scanning the `.text` segment for the 128-bit `call` instructions and decoding their target addresses ([33:80]) [25], we are able to identify the addresses of dynamically linked CUDA library functions.
- **Heap, global stack, user-declared constant memory:** For those user-accessible segments, we insert a known constant flag such as `0xdeadbeef` in kernel code. We then search the physical memory for this flag and resolve the associated virtual addresses using the GPU page tables.
- **CPU-side memory mapped by GPU:** We identify these regions in three steps. First, we parse GPU page tables to extract their physical mappings (which correspond to intermediate physical addresses when the IOMMU is enabled). Second, we traverse the CPU and IOMMU page tables to recover the actual CPU virtual addresses. Finally, we correlate these addresses with `/proc/<pid>/maps` entries to infer their semantic purpose.
- **Other segments:** Remaining unclassified memory regions are examined manually. Through raw binary inspection, we recognize structures such as string tables, function tables, and process-shared memory regions.

This flag-guided probing approach allows us to reliably

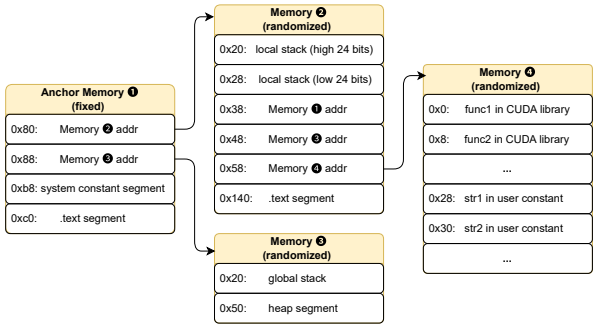


Figure 2: AnchorTrace technique for collecting different randomized segment address. The memory ❶ is a fixed address, so we can use it as an anchor to trace other randomized addresses. The offset refers to the relative offset within the corresponding memory region. All addresses are 64-bit aligned.

identify even hidden or undocumented memory segments, achieving a comprehensive semantic mapping of GPU virtual memory layout that is unknown beforehand. For completeness, We document a set of failed reverse-engineering attempts and more details in the appendix appendix A. Although not directly contributing to our final technique, these cases may offer insights and practical guidance for future efforts.

3.2. AnchorTrace: Recursive Pointer Tracing from Anchor Memory to Collect Randomized Addresses

After recovering the GPU virtual memory layout in a non-randomized setting using FlagProbe described in 3.1, we proceed to analyze how this layout is transformed under ASLR. Specifically, we aim to systematically analyze how the GPU ASLR affects the layout of memory segments. This includes examining whether ASLR changes the relative ordering of segments, whether it applies coarse- or fine-grained randomization, and how much entropy it introduces into each memory region. To conduct this analysis, we must collect a large number of runtime addresses from repeated executions of the same program. On the CPU side, prior work typically executes the same binary millions of times [5], using in-program `printf` statements to expose memory segment addresses directly during each run. These addresses are then collected and analyzed statistically to evaluate the randomness and entropy of each segment. Applying the same methodology to NVIDIA GPUs, however, introduces several unique challenges.

Challenge 2. It is highly challenging to efficiently collect memory segment addresses across repeated GPU executions. First, many GPU memory segments are inaccessible through conventional CUDA kernel code (by printing the address of a variable/function with `printf`). As discussed in §3.1, some segments such as `system constant memory` and `CUDA library` are entirely hidden from user code. Although we can reverse-engineer and locate these segments using FlagProbe, we cannot print their base addresses at runtime during each execution. On the other hand, addresses printed by `printf` in CUDA kernel code are also unreliable, such as the `local/global stack` addresses.

Another approach is to extract segment addresses from the GPU page tables during every sampling, but this method is prohibitively inefficient in practice. CPUs expose dedicated registers (e.g., CR3) that point to the root of their page tables. In contrast, NVIDIA GPUs either do not expose such information or keep it undocumented. In addition, the physical location of the GPU page table root varies across program executions. As a result, we must dump the entire GPU physical memory (24 GB on an RTX 4090), scan it page by page to locate the page table pages, and walk the hierarchy to reconstruct the address mappings. This process takes over one minute per sample, while analyzing ASLR entropy requires collecting more than 100,000 such samples, totaling over 1,666 hours of processing time. Furthermore, launching each CUDA program also incurs significant overhead due to the slow GPU context initialization and cross-processor communication. As a result, per-run sampling is significantly slower on GPUs than on CPUs, which makes large-scale address collection impractical on GPUs.

Our technique. To address these limitations, we propose AnchorTrace, a pointer-chasing-based technique tailored for collecting randomized GPU addresses efficiently. The key idea is to start from a known, non-randomized anchor region and recursively traverse memory by following pointer values stored within each region. This turns the problem into identifying memory regions that contain valuable pointers and segment addresses.

We begin with a one-time reverse-engineering effort, using GPU page tables and memory dumps, to identify all valuable memory regions and the offsets inside them that contain pointers. Since ASLR only shifts the base address of a memory block while preserving the internal pointer offsets, we performed this reverse-engineering in a `cuda-gdb` environment where ASLR is disabled. To reliably extract pointers from raw memory, we rely on two features: pointers are typically 8-byte aligned and often carry the prefix `0x7fff`.

We choose the `system constant memory` as our reverse-engineering starting point, because we find it contains several pointer values that are unique within physical memory during the FlagProbe analysis. For example, offset `0x140` holds the CUDA kernel entry point address in the `.text` segment, as well as pointers to other memory regions. Once the initial regions were identified, we apply both forward and backward pointer-chasing strategies. The forward approach identifies pointers contained within the current memory region and follows them to discover additional regions they reference. Conversely, the backward approach starts from the virtual address of the current region and scans the raw memory dump for pointer values matching this address, thereby locating more other regions that contain references to it. Together, these complementary methods progressively construct a *memory graph*, where nodes represent memory regions and edges denote pointer relationships, as illustrated in Figure 2.

TABLE 1: Overview of experiment settings we test.

GPU	OS	CUDA	Driver
3070Ti	Ubuntu 20.04	11.8	560.35.03
3070Ti	Ubuntu 20.04	12.1	560.35.03
4090	Ubuntu 22.04	12.1	560.35.03
4090	Ubuntu 22.04	12.6	560.35.03
4090 laptop	Ubuntu 24.04	12.6	560.35.03
4090 laptop	Ubuntu 24.04	12.6	570.133.07

A key insight is that although most memory regions are randomized under ASLR, a specific region (memory ❶) remains fixed across executions [1] and can be served as a reliable anchor for pointer chasing, as illustrated by solid arrows in Figure 2. For instance, from this anchor (memory ❶), we can directly retrieve the addresses of the `system constant memory` and `.text` segments at offsets `0xb8` and `0xc0`, respectively. The pointer at offset `0x88` leads to another region that contains pointers to the `global stack` and `heap` segments. Consequently, using this memory graph, we can recursively retrieve the addresses of all memory segments. We implement this pointer-chasing logic in a custom CUDA kernel, which is repeatedly executed to extract randomized segment addresses at runtime. Since all offsets and pointers are known, each kernel execution simply follows a fixed dereference sequence. For example, suppose a fixed anchor region sits at address `0x20000000`. The kernel first reads an 8-byte value at `0x20000080` (using the `printf` statement) to obtain the address of memory ❷ (e.g., `0x7fffaaaa1000`). It then reads the 8-byte value at `0x7fffaaaa1000 + 0x58` to obtain the address of memory ❹. The kernel repeats this dereference chain recursively, following the known offsets until all target segment addresses are resolved. In this way, no additional memory dumps or runtime reverse-engineering are required.

Instead of exhaustively scanning GPU physical memory or reconstructing full page tables for every sample, AnchorTrace selectively inspects a small set of critical regions and extracts specific offsets within them to recover all relevant segment addresses. This design significantly reduces collection overhead. In our experiments, AnchorTrace executes 100,000 instances of the collection kernel and recovers all segment addresses in just 9 hours. This achieves a 185× speedup over page-table-based extraction, which takes 1666 hours.

3.3. Implementation

Leveraging our two proposed techniques: FlagProbe and AnchorTrace, we design and implement a framework to systematically analyze ASLR behavior on NVIDIA GPUs. It consists of three main steps.

Step 1: Dumping GPU memory and extracting GPU page tables. We begin by executing a CUDA program under `cuda-gdb` after disabling ASLR and setting a breakpoint at the kernel's return point to suspend execution. At the breakpoint, we use the `Dumper` tool from prior work [49] to dump the entire GPU physical memory. The resulting memory dump includes GPU page tables and other critical data structures. We then extract the page tables from the memory dump. The existing tool `extractor` [1] can parse page tables on earlier architectures, but it fails to handle certain mappings such as GPU-mapped CPU physical memory, and does not support the Ada architecture (e.g., RTX 4090). To address these limitations, we extend the tool with over 1,600 lines of additional code. This enhanced tool allows us to extract a frozen snapshot of the GPU page tables from the physical memory dump.

Step 2: Identifying memory segment semantics. We next reconstruct a high-level view of the GPU virtual memory layout from dumped page tables. By aggregating all page table entries, we merge contiguous regions with identical permissions into unified memory segments—an approach inspired by the Linux kernel's `vm_area_struct` [9]. This abstraction simplifies subsequent analysis. However, the resulting layout lacks information about the type of content (e.g., text, heap) mapped to each region. We apply FlagProbe to identify the semantics of each segment and uncover hidden memory regions such as `system constant memory` and `CUDA library` segments. This semantic understanding serves as a necessary baseline for analyzing how ASLR transforms the memory layout.

Step 3: Analyzing ASLR Behavior. To investigate how ASLR is implemented, we leverage AnchorTrace to efficiently collect the addresses of key memory segments across multiple executions. Specifically, we identify critical memory regions (as shown in Figure 2), and develop a CUDA kernel that traverses pointer chains starting from the fixed memory region (❶) to trace the locations of other randomized regions (❷, ❸ and ❹). From these four regions, we extract addresses of different segments using identified offsets. We execute the kernel 100,000 times and collect the full set of randomized addresses in each run. This large-scale dataset allows us to perform a statistical analysis of the address distributions for each segment. We use this analysis to characterize ASLR behavior, estimate entropy, and identify potential weaknesses.

We will introduce our analysis results in the following sections. Unless otherwise specified, all addresses presented in this paper are obtained from experiments on an NVIDIA GeForce RTX 4090 laptop GPU (Ada architecture). The experiments are conducted on Ubuntu 24.04 with Linux kernel version 6.11.0. The software environment includes CUDA 12.6, NVIDIA's open-source driver version 560.35.03, and `glibc` 2.39. The hardware consists of a 4090 laptop GPU with 16 GB of VRAM, an Intel core i9-14900HX processor, and 32 GB of CPU memory. Moreover, we also conduct experiments on different GPUs and CUDA versions, and all the findings are reproduced, as shown in Table 1.

1. We initially tried to collect randomized addresses using the page table approach. Although we only collected 100 times, which is far from the number necessary for entropy analysis, we still find that there is a segment in the lower address space that are not randomized.

TABLE 2: Overview of the complete GPU memory layout without ASLR. These results are based on experiments conducted on an NVIDIA RTX 4090 GPU 24GB with CUDA version 12.6. `SYS` means that it maps CPU physical memory. `VID` means that it maps GPU physical memory. The * means the GPU and CPU have the same virtual address that maps the same GPU/CPU physical address.

	Range	Type	Region name	Description
non-randomized regions	0x200000000 - 0x200400000	VID	unknown	unknown data structure
	0x200400000 - 0x200600000*	VID	/dev/nvidia0	NVIDIA device file region for GPU #0
	0x200600000 - 0x203600000*	SYS	/dev/nvidiactl	/dev/nvidiactl from CPU physical memory
	0x203600000 - 0x205c80000	VID	process shared memory	physically shared by all GPU processes
	0x205c80000 - 0x205c90000	SYS	64 KB kernel page	CPU mem. shared by all GPU processes
	0x205c90000 - 0x206009000	VID	process shared memory	physically shared by all GPU processes
	0x206400000 - 0x206800000*	SYS	/dev/nvidiactl	/dev/nvidiactl from CPU physical memory
	0x206800000 - 0x206a00000*	SYS	/dev/nvidia-uvm	/dev/nvidia-uvm from CPU physical memory
	0x206a00000 - 0x206c00000	VID	unknown	contains pointers to sensitive regions
	0x206c00000 - 0x206e00000	SYS	/dev/zero	this region maps to CPU VA 0x7ffff0a00000
	0x206e00000 - 0x207a00000	VID	heap	heap space for CUDA GPU runtime
	0x10000000000 - 0x10000200000	VID	all zero region	all bytes are zero
	0x10002000000 - 0x10002200000	VID	all zero region	all bytes are zero
rondomized regions	0x7fffa2000000 - 0x7fffbac00000	VID	global stack	actual stack region for CUDA GPU runtime
	0x7fffbac00000 - 0x7fffbae00000*	SYS	/dev/zero	/dev/zero from CPU physical memory
	0x7fffbae00000 - 0x7fffbb000000	VID	all zero region	all bytes are zero structure
	0x7fffbb000000 - 0x7fffbb601000*	SYS	/dev/zero	/dev/zero from CPU physical memory
	0x7fffcec00000 - 0x7fffcee00000	VID	user-declared constant	strings & __constant__ variables defined in kernel
	0x7fffd0000000 - 0x7fffd3c00000	VID	unknown	unknown data structure, contains heap pointers
	0x7fffd4000000 - 0x7fffd4400000	VID	system constant memory	CUDA internal constant memory
	0x7fffd4400000 - 0x7fffd4a00000*	SYS	/dev/nvidiactl	/dev/nvidiactl from CPU physical memory
	0x7fffd4a00000 - 0x7fffd4c00000	VID	func and string table	CUDA library function ptr and string ptr table
	0x7fffd4e00000 - 0x7fffd5000000	VID	CUDA library	CUDA library code for GPU runtime
	0x7fffd5000000 - 0x7fffd5200000*	SYS	/dev/nvidiactl	/dev/nvidiactl from CPU physical memory
	0x7fffd5200000 - 0x7fffd5400000	VID	.text	user-provided CUDA kernel code
	0x7fffd5400000 - 0x7fffd5600000*	SYS	/dev/zero	/dev/zero from CPU physical memory
	0x7fffd5600000 - 0x7fffd5a63000*	SYS	/dev/nvidiactl	/dev/nvidiactl from CPU physical memory

4. Comprehensive GPU Memory Layout

In this section, we conduct a detailed analysis of the GPU virtual memory layout using the reverse engineering technique introduced in §3.1. We first reconstruct the GPU process address space in §4.1, identify the semantics of each memory region, and uncover several undocumented segments. Next, in §4.2, we examine the relationship between GPU and CPU memory layouts, highlight architectural correlations, and show that several memory regions can be exploited across processor boundaries.

4.1. Memory Layout on GPU

Using the FlagProbe technique described in §3.1, we systematically reverse-engineer the semantics of memory segments in the GPU's virtual address space. Table 2 summarizes the results, presenting the address ranges, content types, and corresponding functionalities of each segment. To our knowledge, this is the first comprehensive view of the GPU process memory layout.

Specifically, the GPU's virtual memory layout comprises several critical segments. The `.text` region contains the compiled CUDA kernel binary. The `CUDA library` provides important internal runtime functions, such as a CUDA-specific version of `printf`, which differs from the one in `glibc`. Additionally, the `local stack` and `heap` segments support thread-local execution contexts, while the `global stack` stores the concatenated `local stacks` of the 32 threads in a warp. Each thread allocates memory in 8-byte units [34]. This means that the first 32*8 bytes of `global stack` store the first 8 bytes `local stack` for 32 threads within a warp and so on. Moreover, the layout also includes mappings for critical metadata, such as `function & string table` and `system constant memory`. The layout incorporates mappings to CPU-side device memory (system memory) as well. These mappings include shared regions such as `/dev/nvidia-uvm`, `/dev/zero`, and `/dev/nvidiactl`, which facilitate interaction between the two processors. In addition, there are also three unknown regions we cannot determine the functions since they contain only a few numbers of values like `0x390100` and pointers.

Among all the GPU-mapped regions, we identify two memory regions with unusual properties that may pose security risks. These regions are shared across all running CUDA processes, making them susceptible to covert channel and code injection attacks. We find that two memory regions, each approximately 42MB in size and located at `0x203600000-0x205c80000` and `0x205c900000-0x206009000`, are shared across all CUDA kernel processes. As shown in the "process shared memory" entry in Table 2, these regions map the same physical memory to identical virtual addresses across processes. They are configured with `RWX` permission and exhibit volatile behavior. Any write operation to these regions modifies their contents, and these changes persist even after the corresponding CUDA process terminates. This persistence enables potential attack vectors like covert

channel attacks and code injection exploits. For instance, one process can inject malicious GPU binary code into these fixed-address regions, and another process can later execute it by exploiting vulnerabilities to hijack control flow.

> **Finding-1:** All CUDA processes share two large memory regions with `RWX` permission (marked as `process shared memory` in Table 2), which have identical GPU virtual and physical addresses. This facilitates covert channel and code injection attacks.

Another notable finding is that a 64KB memory region mapped to CPU physical memory is shared across all CUDA processes. The range of this region is from `0x205c80000` to `0x205c90000`, as named `64 KB kernel page` in Table 2. Both its GPU virtual address and CPU physical address remain consistent across different CUDA kernel executions. More critically, this physical memory is not mapped into user space on the CPU. Instead, it resides in kernel space. This design raises a significant security concern: CUDA programs executing with user-level privileges on the GPU are able to directly read from and write to kernel pages on the CPU. Such access potentially bypasses conventional system-level access control mechanisms. Although the exact purpose of this 64KB region remains unclear, binary inspection reveals the presence of structured binary data. Moreover, we further confirmed through experiments that overwriting this region within CUDA kernels does not cause program crashes or trigger any observable execution errors. This observation suggests that the region does not store control data. We hypothesize that it serves as an internal data exchange buffer between the GPU and the CPU.

> **Finding-2:** All CUDA processes run with user-level privileges on the GPU but map and share the same 64KB memory page located in CPU kernel space (marked as the `64 KB kernel page` in Table 2). Such mapping potentially bypasses system-level protection mechanisms and results in unauthorized memory accesses.

4.2. Relationships Between GPU and CPU Memory Mappings

We analyze the interactions between CPU and GPU memory by comparing the GPU address space we reverse-engineered with the CPU-side virtual memory layout. The CPU memory layout is obtained from `/proc/<pid>/maps`, and the correlation is shown in Figure 3.

CUDA applications include additional memory regions in their CPU-side address space that are not present in traditional CPU-only programs. These regions include memory-mapped device files such as `/dev/nvidiactl` and `/dev/nvidia-uvm`. The GPU driver uses these virtual files for device control and unified memory management. On the

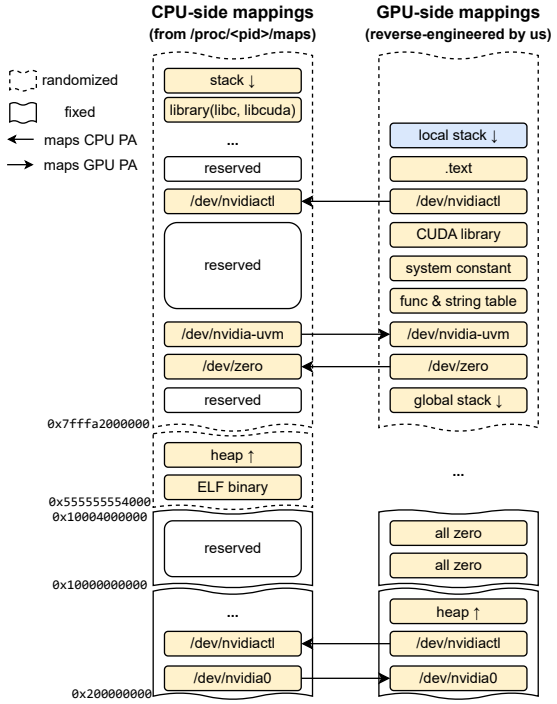


Figure 3: CPU and GPU memory layouts without ASLR and their correlation. The left and right sides show their virtual mappings; aligned regions share virtual addresses. Yellow boxes represent regions backed by physical memory. The white boxes (reserved) are unmapped, meaning they are assigned virtual addresses but do not have physical memory backing.

CPU side, these regions typically appear with `RW-S` permissions, indicating readable, writable, and shared mappings. Interestingly, we observe that the physical memory pages backing these CPU regions are also mapped into the GPU virtual address space. What's more, these mappings use the exact same virtual addresses on both the CPU and GPU sides. This implies a strong correlation between the CPU and GPU virtual addresses[2].

In addition, we find a number of anonymous virtual memory regions (marked as `reserved` in Figure 3) in the CPU address space that are marked with permission `---p`, suggesting that they reserve virtual address ranges without being backed by physical memory. However, we discover that these virtual addresses also appear in the GPU page tables with valid mappings to GPU physical memory. This observation provides further evidence of a coordinated virtual address design between CPU and GPU, and it inspired our later investigation into whether GPU ASLR might be influenced by CPU-side ASLR.

We also identify a GPU physical memory region that is mapped into both CPU and GPU virtual address

2. This has also inspired us to explore the correlation between GPU-side ASLR and CPU-side ASLR.

spaces using identical virtual addresses. This region, named `/dev/nvidia0` in Figure 3, spans 2MB from `0x200400000` to `0x200600000`. On the CPU side, this region is mapped with standard `RW-S` permissions and on the GPU side it is mapped with `RW` permissions. However, the GPU page table format lacks an explicit execute (X) bit [7], so the same region is implicitly executable on the GPU side. This permission asymmetry creates a potential security risk. For example, attackers could write GPU-executable payloads through the CPU's writable mapping to help GPU code injection attacks. Alternatively, they may craft controlled data such as compromised stacks from the GPU side to aid CPU-side vulnerability exploitations.

> **Finding-3:** The CPU and GPU share portions of their virtual addresses, reflecting a strongly correlated address space design. Some shared regions, such as `/dev/nvidia0` in Table 2, present inconsistent permissions. These regions allow code injection into GPU space from the CPU side and data injection from the GPU into CPU-accessible memory, creating cross-processor attack surfaces.

5. GPU Address Space Layout Randomization

In this section, we present a detailed analysis of GPU ASLR using the technique introduced in §3.2. Our goal is to evaluate the randomness and effectiveness of GPU ASLR while investigating its security weaknesses. To achieve this, we treat the collected randomized addresses as discrete data points and systematically analyze their randomness using well-defined metrics. We first introduce the evaluation metrics in §5.1. These metrics allow us to measure the randomness of both individual memory segments and their inter-dependencies. Next, we present our findings on GPU ASLR implementation in §5.2, detailing how randomization is applied to GPU memory layouts. Finally, we investigate the relationship between GPU and CPU ASLR in §5.3.

5.1. Analysis Targets and Metrics

Targets. To evaluate the implementation and effectiveness of GPU ASLR, we focus on memory segments critical to kernel execution and potential exploitation, specifically code and data regions such as the heap, stack, and `.text`. Although the GPU address space includes mappings of CPU device memory, we exclude them from our analysis as they do not directly affect kernel logic. To further assess cross-processor ASLR behavior, we also examine corresponding CPU-side memory regions—including stack, heap, and shared libraries—to identify address correlations and potential security weaknesses in memory management between CPU and GPU.

Metrics. To evaluate the effectiveness of GPU ASLR, we adopt entropy-based metrics grounded in information theory [24]. While traditional metrics like Shannon entropy [43] or NSB entropy [32] quantify randomness in general systems, they offer limited insight into the specific behavior of address space randomization. Therefore, we employ two specialized metrics inspired by prior ASLR research [5] to evaluate the randomness: *Absolute Entropy* and *Correlation Entropy*.

The *Absolute Entropy* quantifies the degree of randomization for a single memory object by measuring which bits of its address vary across runs. This metric highlights the granularity of the randomization. For example, if a memory region's address changes between bit positions 26 and 44, its *Absolute Entropy* is 19 bits, indicating a randomization granularity of 2^{26}. Furthermore, if another memory region exhibits the same bit variation range (also [26:44]), it suggests that both regions are randomized within the same address block, revealing possible alignment in the memory layout.

Formally, let $a^{(n)}$ be the n-th observed address and $a_i^{(n)}$ its i-th bit. The *Absolute Entropy* is defined as:

$$H_{\text{abs}}(a) = \sum_{i=1}^{m} \delta_i, \quad \delta_i = \begin{cases} 1, & \text{if } \exists\, n, n' : a_i^{(n)} \neq a_i^{(n')} \\ 0, & \text{otherwise} \end{cases}$$

where m is the address bit width. This metric captures both the range of address variation and the granularity of randomization.

The *Correlation Entropy*, on the other hand, assesses whether two memory regions—within the GPU or across CPU and GPU—maintain a fixed spatial relationship across runs, which could indicate structural leakage or ASLR weaknesses. Rather than directly comparing the raw bitwise differences between addresses, which may be influenced by the absolute value and thus mislead the analysis, we focus on the entropy of their offsets. For instance, consider two addresses fixed at `0xff00` and `0x00ff` in every randomization. Measuring changed bits directly would suggest a 16-bit difference, failing to capture the constant relationship between them. However, computing the offset between these addresses yields a constant value `0xff00 - 0x00ff = 0xfe01`, indicating a fixed correlation despite address randomization.

Specifically, let $x^{(n)}$ and $y^{(n)}$ be the n-th observed addresses of two memory objects, and define the offset as $o^{(n)} = x^{(n)} - y^{(n)}$. Then, the *Correlation Entropy* $H_{\text{corr}}(x, y)$ is defined as the *Absolute Entropy* of the offset:

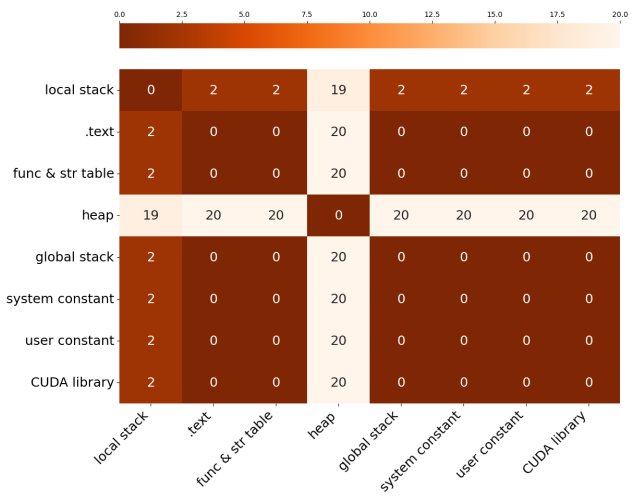
$$H_{\text{corr}}(x, y) = H_{\text{abs}}(o).$$

This metric accurately captures the randomness of the relative positions between the two regions.

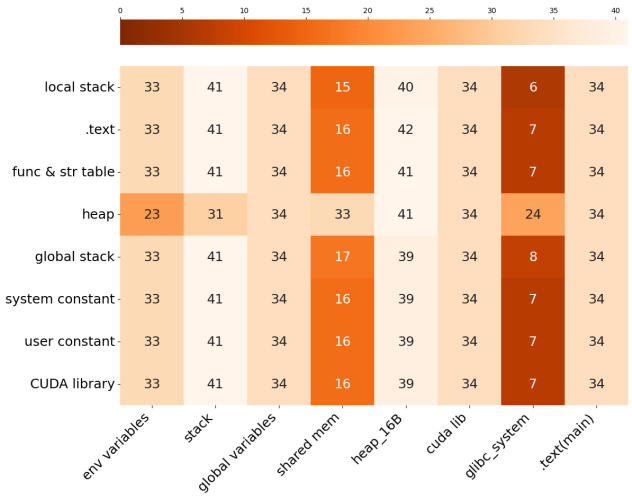
5.2. Absolute Entropy

We analyze the *Absolute Entropy* (H_{abs}) and the bitwise variation for various memory objects in both GPU and CPU address spaces. The results are summarized in Table 3.

GPU-side objects. On the GPU, most memory objects except the `heap` and local stack exhibit consistent entropy values of 20 bits, with changed bits spanning positions 25



(a) The correlation entropy between different GPU objects. Both the X-axis and Y-axis represent GPU objects.



(b) The correlation entropy between different CPU and GPU objects. The X-axis represents CPU objects, and the Y-axis represents GPU objects.

Figure 4: Correlation entropy results of GPU ASLR. Smaller values are depicted with darker colors, indicating a lower degree of randomization that makes it easier for one object to predict another.

TABLE 3: Absolute Entropy (H_{abs}) and changed bit range for various CPU and GPU memory regions. The H_{abs} means the number of changed bits of an object with randomization.

	Memory object	H_{abs}	Changed Bits
GPU	local stack	19	[26, 44]
	global stack	20	[25, 44]
	.text	20	[25, 44]
	system constant mem	20	[25, 44]
	heap	0	None
	user-declared constant	20	[25, 44]
	func & string table	20	[25, 44]
	GPU library	20	[25, 44]
CPU	env variables	22	[12, 33]
	stack	30	[4, 33]
	global variables	34	[12, 45]
	shared memory	33	[12, 44]
	heap (16B size)	41	4, [6, 45]
	CUDA library	34	[12, 45]
	glibc (system)	24	[21, 44]
	.text (main)	34	[12, 45]

to 44. This implies that GPU ASLR is applied with a coarse granularity of 2^{25}. Notably, the `heap` region is an exception, showing zero entropy: its address remains fixed across all executions. This indicates that the heap is completely deterministic, lacking any address randomization.

CPU-side objects. In contrast, CPU-side objects demonstrate significantly higher entropy and broader ranges of bit variation. For example, the address of `glibc_system` exhibits 24 bits of entropy (bit 21 to 44), and the `stack` reaches 30 bits (bit 4 to 33). Even the region with the lowest entropy (i.e. `env variables`) still achieves 22 bits, exceeding the entropy of all GPU regions.

As nearly all objects vary across exactly the same bit range [25, 44], we hypothesize that these memory objects are randomized together as part of a single block. This may raise issues that the random address used for one object may implicitly determine the layout of the others. However, *Absolute Entropy* alone cannot verify this hypothesis, as it only reflects per-object variation. We substantiate this hypothesis in §5.3 using *Correlation Entropy*.

> **Finding-4:** GPU heap memory exhibits no address randomization, while other GPU regions have limited entropy (20 bits) and identical changed-bit ranges. This suggests a coarse randomization strategy on the GPU, which makes it easier for attackers to predict.

5.3. Correlation Entropy

We analyze the *Correlation Entropy* (H_{corr}) of different GPU memory objects as well as the H_{corr} between GPU and CPU memory objects. The result is presented in Figure 4.

GPU and GPU objects correlation. As shown in the left part of the Figure 4a, most GPU memory objects exhibit zero entropy in their offsets relative to one another, except for the `heap` and `local stack`. This demonstrates that GPU memory randomization is highly deterministic for most objects, which remain fixed across executions. Moreover, this indicates that all GPU memory regions, except the `heap` and `local stack`, share the same randomization offset. As a result, if any single address is leaked, the remaining addresses can be reliably inferred using their relative offsets.

> **Finding-5:** The GPU ASLR is collectively randomized, with most memory regions sharing the same randomization offset. This design allows attackers to infer the addresses of most other memory regions once a single address is leaked.

GPU and CPU objects correlation. The right part of the Figure 4b reveals the correlation entropy between GPU and CPU memory objects. Most GPU-CPU memory object pairs show high entropy values exceeding 33 bits, suggesting strong randomization and low correlation. However, a notable exception is observed with the `glibc` memory region on the CPU. The correlation entropy between `glibc` and most GPU memory objects (excluding the `heap`) is as low as 7 bits. This low entropy indicates a strong correlation between these addresses, enabling attackers to predict the `glibc` addresses on the CPU using leaked GPU addresses. Since `glibc` contains critical functions (e.g., `system`) and exploitable ROP gadgets, this predictability poses a significant security risk.

> **Finding-6:** The offsets between CPU `glibc` and GPU memory objects change by as little as 7 bits during randomization. This strong correlation allows attackers to predict `glibc` addresses on the CPU using leaked GPU addresses.

6. Exploiting GPU ASLR

Previous sections reveal weaknesses in NVIDIA GPU ASLR. This section demonstrates how these weaknesses can be exploited to break the GPU and CPU boundary and leak sensitive CPU addresses. In the following, we first define the threat model, outlining the attacker's goal and capabilities in §6.1. Next, we detail our approach to bypass GPU ASLR and further bypass CPU ASLR in §6.2. Finally, we present a case study by using an example CUDA program, showing that it is practical to guess the `system` function address exploiting GPU-CPU ASLR correlation in §6.3.

6.1. Threat Model

Attacker goal. We assume the attacker is an unprivileged user of an AI model inference service [14] deployed on a server equipped with NVIDIA GPUs. There is an OOB read memory vulnerability in the underlying CUDA kernel implementations. These OOB bugs are common in machine learning frameworks [15], [47], [22]. The attacker aims to break GPU ASLR reliably by leaking sensitive address information from the GPU via a common OOB bug in CUDA kernel. The primary goal is to reliably defeat GPU-side ASLR by leaking sensitive GPU virtual addresses via the OOB defect. Using leaked GPU addresses, the attacker may further infer critical CPU-side addresses (e.g., the `glibc` base) despite low GPU–CPU address correlation, thereby obtaining the address information necessary to enable conventional CPU-side exploits like return-oriented programming (ROP). We do not assume a complete CPU exploitation chain, since address leakage is treated as a primitive that enables subsequent CPU attacks, whose concrete construction depends on CPU-side vulnerabilities. The CPU side attack methods have been extensively explored in previous work and lie outside this paper's scope.

Attacker capabilities. We assume the attacker can interact with the deployed AI model via a standard inference API by repeatedly sending crafted inputs and observing the corresponding outputs. This aligns with practical inference-as-a-service scenarios, where users submit inputs (e.g., images or text) and receive model outputs [4]. We further assume the OOB read vulnerability in the CUDA backend can be triggered by malformed inputs. For example, a carefully constructed image with extreme dimensions may exploit insufficient input validation in CUDA image processing kernels [17], [35]. Moreover, the attacker can launch multiple queries to the model, and unless a crash occurs, all queries are processed within the same long-lived GPU process. If a crash is triggered, we assume the server includes an auto-restart mechanism [40] that reinitializes the model service (e.g., by restarting the process). This allows the attacker to perform multiple attempts.

System defenses. On the CPU side, we assume that common system-level defenses are enabled, including stack canaries, Write XOR Execute (W^X), Position-Independent Executables (PIE), and other default protections. Specifically, the system is configured with `randomize_va_space=2`, which corresponds to full ASLR as used by default in Ubuntu and other modern Linux distributions. These defenses make direct code injection or ROP-style attacks infeasible unless address space information is first leaked. On the GPU side, we assume that NVIDIA's GPU ASLR mechanism is active. However, due to its limited entropy and incomplete randomization coverage, GPU ASLR provides weaker protection in practice. As such, the attacker targets the GPU as the accessible entry point, leveraging GPU vulnerabilities to break inter-processor isolation and subsequently leak CPU-side address information, which is a challenging task from the CPU side alone in modern systems.

6.2. Bypassing ASLR

To break the GPU-side ASLR and subsequently the isolation between the GPU and CPU, the attacker must first leak a randomized address within the GPU memory space. Leveraging this leaked address and Finding-6 (which reveals that the entropy between GPU and CPU virtual address spaces is limited to only 7 bits), the attacker can infer the corresponding CPU address (e.g., `glibc` base). This section details how the attacker achieves these two steps, i.e., bypassing GPU and CPU ASLR.

Bypassing GPU ASLR. While we have presented the limitations of GPU ASLR, leaking a randomized address within the GPU address space remains non-trivial in practice. First, traditional OOB stack operations cannot access return addresses: due to aggressive compiler optimizations, CUDA kernels compiled without debugging information (i.e., without `-G -g`) are heavily inlined. As a result, user-defined functions are flattened into a single kernel body, eliminating conventional function calls and their corresponding stack frames. Consequently, there are no return addresses pushed onto the stack during kernel execution. Second, although user code may invoke CUDA library functions

such as `printf`, these calls do not store return addresses on the stack either. Instead, return addresses are held in specific registers (e.g., `R20`, `R21`) according to NVIDIA's internal calling convention. Third, OOB reads from other user-accessible memory regions (e.g., heap or user-declared constant memory) are generally ineffective for leaking stable randomized addresses, as the memory areas containing such sensitive information (regions in Figure 2) are far away from these user-controlled regions.

To overcome these limitations, we reverse-engineered the CUDA memory layout and discovered a hidden stack frame located prior to the regular kernel function stack, suggesting the presence of undocumented initialization logic executed before the user-defined CUDA kernel begins. Although this hidden frame does not store return addresses, it exhibits a structured layout in which several fields at fixed offsets contain pointers to the randomized `.text` and `CUDA library` segments. Through differential testing, we observed that the number of bytes required to reach this hidden frame remains constant for a given NVIDIA driver version, and is independent of the CUDA version or GPU model. This consistent layout allows attackers to craft a malicious payload that performs a controlled stack OOB read, reliably disclosing pointers within the randomized GPU address space. As a result, we are able to effectively bypass GPU ASLR.

> **Finding-7:** We discovered a hidden stack frame preceding the regular CUDA kernel stack that contains pointers to randomized code segments such as the `.text` and `CUDA library`. This frame enables reliable leakage of GPU addresses via a controlled stack-based OOB read, thereby bypassing GPU ASLR.

Bypassing CPU ASLR. Once a GPU-side address (e.g., within the `.text` segment) is obtained, the attacker can exploit the limited correlation entropy between GPU and CPU addresses to infer sensitive CPU-side addresses. As shown in §5.3, our analysis reveals that the offset between GPU `.text` objects and CPU-side `glibc` objects exhibits only 7 bits of entropy. Thus, the attacker's challenge reduces to correctly identifying the offset.

Specifically, we observe that the offset between GPU-side `.text` base address and CPU-side `glibc` base address varies across different GPU and driver versions. However, this variation is confined to bits [20:27], while the higher and lower bits remain stable. Furthermore, within the same GPU and driver configuration, ASLR randomization introduces variability only in bits [21:27] of the offset. These two sources of variability partially overlap, and can be jointly modeled as 8 bits of entropy. Consequently, the attacker only needs to randomize bits [20:27] in the offset `0x0XX00000(Anonymized)`, which has a $1/256$ chance of being correct. We consider a per-trial success probability of $1/256$ to be practically exploitable under the parallelism available on modern processors. Prior work [5] has shown that even a per-trial probability as low as $1/2^{19}$ can be overcome in roughly 33 minutes with parallelism.

```
1  __global__ void transform_tensor(uint64_t* input, uint64_t* output,
   ↪  int width) {
2      int row = blockIdx.x * blockDim.x + threadIdx.x;
3      if (row > width)
4          return;
5      ...
6      // temp located at local stack
7      uint64_t temp[512];
8      temp[0] = input[row];
9      for (int i = 1; i < 512; i++){
10         temp[i] = input[i - 1] * row + temp[i - 1];
11         ...
12     }
13
14     // row > 512 will overflow
15     output[row] = temp[row];
16 }
```

Figure 5: An example CUDA program containing an OOB vulnerability. If the input `width` exceeds 512, accessing `temp[row]` will result in an out-of-bounds read.

6.3. Case Study

To demonstrate the practicality of our attacks, we construct a realistic scenario based on a vulnerable CUDA kernel, shown in Figure 5. We embed this kernel into a custom TensorFlow operator to simulate a real-world ML deployment, where users upload and process images through GPU acceleration. The kernel, `transform_tensor`, performs a transformation over a 1D tensor representing image rows. It uses a local stack-allocated array `temp[512]` for intermediate computation and writes the result to `output`. However, if the input tensor width exceeds 512, the variable `row` can exceed the bounds of the `temp` array, resulting in OOB stack read on line 15. This read discloses adjacent stack data, which, as we demonstrate, can include sensitive addresses within the GPU's virtual address space. We demonstrate how an attacker conducts the address leakage attack in the following four steps:

Step 1: Triggering the vulnerability with malicious input. The attacker provides a crafted input, which is an image with a width larger than 512. The image will be converted into a tensor and passed to the CUDA kernel. The kernel processes this input in parallel across CUDA threads. Because the kernel checks only whether `row > width` (line 4) and not whether `row < 512`, threads with `row > 512` can read past the bounds of the local array `temp`, accessing adjacent stack memory. The OOB data is then written to the output tensor and returned to the attacker.

Step 2: Extracting GPU virtual addresses from output. From the returned output tensor, the attacker scans for values resembling valid addresses. For example, a value such as `0x7fffd4e0d6b0` may correspond to a pointer into the GPU-side `CUDA library` segment. Since this segment is aligned to 2MB (verified via page table analysis), the attacker can obtain its base address by zeroing out the lower 20 bits: `cuda_lib_base = 0x7fffd4e00000`.

Step 3: Inferring CPU-side addresses via known offsets. As shown in §6.2, the offset between GPU-side segments

(.text or CUDA library) and CPU-side segments (glibc) is influenced by the GPU model, driver version, and ASLR at runtime. Interestingly, the resulting randomness uniformly affects only bits [20:27] of the offset, with all other bits remaining stable. So the attacker can add the known offset (0x0XX00000(Anonymized)) where only bits [20:27] are randomized: : glibc_base = 0x7fffd4e00000 + 0x0XX00000.

Now the attack can recover the glibc base address. Since only 8 bits are unknown, this guess has a $1/256$ probability of success per attempt. Even if the initial guess is incorrect, the attack can be retried multiple times from Step 1.

7. Discussion and Mitigation

Previous sections presented the weaknesses in GPU ASLR and demonstrated how these weaknesses can be exploited to bypass randomization and launch attacks. In this section, we provide additional discussion and potential mitigation strategies.

7.1. Comparison to Prior Work

Our work corrects two inaccurate conclusions from prior studies. First, Zhang et al. [49] stated that each entry in the PD3-level GPU page table is 1024 bytes, with a page consisting of 4 entries. However, our reverse engineering results reveal that PD3 entries are actually 8 bytes each, consistent with other levels. Only entries 0–3 are valid, while entries 4–511 are all zeroed out. We reproduced their experimental environment and found that some parts of the page tables were missed during scanning, which likely led to the incorrect inference. Second, Guo et al. [17] claimed that the CUDA built-in library segment in GPU virtual memory is not randomized. In contrast, our experiments demonstrate that this segment is indeed randomized, indicating a misunderstanding in their analysis. We discovered that their experiments were conducted under cuda-gdb, which disables ASLR by default and thus may have caused this misunderstanding. We have confirmed these findings with the authors, who agreed with our explanations.

7.2. Mitigation Strategies

Fully randomizing the GPU address space. One of the primary weaknesses of GPU ASLR lies in coarse-grained randomization, as illustrated in Finding-5. To improve the strength of GPU ASLR, we recommend NVIDIA extend randomization to include the CUDA data structure segment. This will prevent the non-randomized addresses from being exploited to leak information and will significantly increase the difficulty of exploiting OOB vulnerabilities.

Enforcing GPU and IOMMU page table permissions. Our analysis revealed that 1) GPU page tables lack the executable bit (X-bit), and 2) GPU accesses to CPU memory through the IOMMU page table lack proper permission enforcement. These allow GPU code to maliciously access and modify the mapped CPU physical memory. Implementing stricter permission controls for GPU and IOMMU page tables can help prevent such unauthorized memory access and modification.

Increasing the entropy of GPU ASLR. GPU address randomization currently provides only 20 bits of entropy, which leaves it vulnerable to brute-force attacks even without memory leaks. Increasing the entropy of GPU address randomization would make brute-forcing impractical. In addition, the offsets between different GPU memory regions should be randomized to enhance Correlation Entropy, making address inference more challenging and reducing the probability of successful attacks.

Updating to the latest NVIDIA driver. NVIDIA has addressed some of these issues in recent driver releases. In particular, the exploitation method described in Finding-7 has been mitigated in version 570 of the driver. We strongly recommend users update to the latest driver versions to benefit from these security improvements. However, if a stronger GPU vulnerability such as arbitrary address read exists, the ASLR bypass can still be triggered by accessing the fixed memory region described in Figure 2.

8. Related Work

ASLR analysis. ASLR has been a cornerstone of modern memory protection mechanisms since it was first introduced by the PaX team in 2003 [45]. Over the years, numerous studies have systematically analyzed ASLR across different platforms. Binosi et al. [5] conducted a comprehensive evaluation of ASLR implementations on Linux, Windows, and macOS, uncovering low-entropy weaknesses that enable rapid bypasses. Similarly, Marco-Gisbert et al. [16] exposed vulnerabilities in ASLR on Linux and PaX systems, subsequently introducing ASLR-NG [27], a framework for systematic ASLR analysis. Jang et al. [19] investigated the paradoxical scenario where, under certain rare conditions, ASLR inadvertently facilitates memory exploitation. They also identified real-world vulnerabilities demonstrating this effect. Koschel et al. [23] demonstrated a side-channel attack that bypasses KASLR. Beyond these efforts, extensive research has explored ASLR weaknesses across various platforms, including Windows 10 [10], IoT firmware [39], and Linux kernel drivers [33].

In contrast to these prior works, which primarily focus on CPU-based systems or well-documented software environments, our study targets NVIDIA GPUs—an area that remains largely unexplored and is often treated as a black box due to the lack of documentation beyond limited open-source driver code.

GPU vulnerability analysis. Research on vulnerabilities on GPUs has primarily focused on microarchitecture analysis, side-channel attacks, and memory vulnerabilities. In the realm of side-channel analysis, Jia et al. analyzed the Tensor Cores and cache hierarchies of NVIDIA Turing [20] and Volta [21] architectures using benchmarking techniques. Abdelkhalik et al. [2] extended similar analysis to NVIDIA Ampere GPUs.

In terms of side-channel attacks, Naghibijouybari et al. [30] introduced a cross-process side-channel attack on GPUs to achieve information leakage. Similarly, Zhang et al. [49] discovered that processes isolated by MIG on NVIDIA Ampere GPUs still share the L3 cache, enabling side-channel attacks. Nayak et al. [31] exploited the last-level cache on Pascal GPUs, while Dutta et al. [12] leveraged the NVLink interconnect between CPU and GPU to exfiltrate data.

Regarding memory vulnerabilities, the work most closely related to ours is by Guo et al. [17], who conducted a systematic analysis of OOB vulnerabilities on NVIDIA GPUs and reverse-engineered the stack memory layout. However, their analysis assumes that ASLR is disabled, whereas our work focuses on ASLR itself and demonstrates how to bypass it. Mittal et al. [29] provided a comprehensive survey of GPU vulnerabilities, categorizing exploit patterns such as data leaks, side- and covert-channels. Miele et al. [28] exploited stack overflows on TITAN GPUs to hijack function pointers, while also analyzing instructions related to return addresses and the feasibility of ROP attacks. Park et al. [37] proposed the "Mind Control" attack, a novel method that manipulates GPU device memory to utilize fixed CUDA library code addresses to disrupt deep learning systems, significantly degrading the model's prediction accuracy to near-random levels. Sorensen et al. [44] introduced the "LeftoverLocals" attack, exploiting uninitialized GPU local memory to recover sensitive data across processes or containers. Their work demonstrated the ability to eavesdrop on other users' interactive LLM sessions by recovering data from the local memory of Apple, Qualcomm, and AMD GPUs. Roels et al. [41] studied ROP gadgets in GPU memory and proposed a "compound gadget" method to bypass NVIDIA's defense of storing return addresses in registers. However, they did not assume ASLR was bypassed, nor did they analyze the implementation details of ASLR, leaving the practical feasibility of such attacks unaddressed.

Compared to these studies, our work expands the scope of GPU security research by focusing on NVIDIA GPU ASLR, a field largely unexplored in existing literature. We not only reveal, for the first time, critical flaws in NVIDIA GPU ASLR but also demonstrate how these weaknesses can be exploited to bypass CPU ASLR, enabling attacks that span both GPU and CPU domains.

9. Conclusion

This paper presents the first comprehensive examination of ASLR on NVIDIA GPUs. By dumping and analyzing GPU page tables, we reveal a detailed GPU memory layout and define metrics to evaluate randomization entropy. Our findings expose previously unknown vulnerabilities, including an unrandomized GPU heap and correlated offsets between GPU and CPU regions, which can undermine both GPU and CPU ASLR. Next, a practical case study demonstrates how these flaws enable attackers to infer CPU addresses from the GPU. Finally, we give mitigations to improve the security of GPU ASLR.

Acknowledgments

The authors would like to thank our shepherd and reviewers for their insightful comments. Those comments helped to reshape this paper. This work is partially supported by the National Natural Science Foundation of China (Grants No. 62572432 and No. 62532012). Dr. Guo is only supported by start-up funding at the University of Rochester.

References

[1] 0x5ec1ab. gpu-tlb/extractor. https://github.com/0x5ec1ab/gpu-tlb/tree/main/extractor, 2025. Accessed December 25, 2024.

[2] Hamdy Abdelkhalik, Yehia Arafa, Nandakishore Santhi, and Abdel-Hameed Badawy. Demystifying the nvidia ampere architecture through microbenchmarking and instruction-level analysis. *arXiv preprint arXiv:2208.11174*, 2022.

[3] Damian Andrysiak. Why variables in thread's block have the same memory address? cuda, 2021. Stack Overflow, Question ID: 66368478.

[4] Tim J. Baek and Open WebUI contributors. Open webui: User-friendly ai interface (supports ollama, openai api, ...). https://github.com/open-webui/open-webui, 2025. Accessed June 6, 2025.

[5] Lorenzo Binosi, Gregorio Barzasi, Michele Carminati, Stefano Zanero, and Mario Polino. The illusion of randomness: An empirical analysis of address space layout randomization implementations. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 1360–1374, 2024.

[6] Cristina Cismaru, Ruxandra Chiroiu Trandafir, and Emil-Ioan Slusanschi. A study of gpu memory vulnerabilities. In *2023 22nd RoEduNet Conference: Networking in Education and Research (RoEduNet)*, pages 1–9. IEEE, 2023.

[7] NVIDIA Corporation. Gp100 mmu format. https://nvidia.github.io/open-gpu-doc/pascal/gp100-mmu-format.pdf, 2024. Accessed December 25, 2024.

[8] NVIDIA Corporation. Open gpu kernel modules. https://github.com/NVIDIA/open-gpu-kernel-modules, 2024. Accessed December 24, 2024.

[9] Linux Kernel Developers. Linux memory management types header file. https://elixir.bootlin.com/linux/v6.12.6/source/include/linux/mm_types.h#L667, 2024. Accessed December 25, 2024.

[10] Raquel Vázquez Díaz, Martiño Rivera-Dourado, Rubén Pérez-Jove, Pilar Vila Avendaño, and José M Vázquez-Naya. Address space layout randomization comparative analysis on windows 10 and ubuntu 18.04 lts. *Engineering Proceedings*, 7(1):26, 2021.

[11] Erica Douglas. Nvidia vs. amd discrete gpu market share (2010 to 2024). https://pcviewed.com/nvidia-vs-amd-discrete-gpu-market-share, 2025. Accessed January 24, 2025.

[12] Sankha Baran Dutta, Hoda Naghibijouybari, Nael Abu-Ghazaleh, Andres Marquez, and Kevin Barker. Leaky buddies: Cross-component covert channels on integrated cpu-gpu systems. In *Proceedings of the 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 15–28. IEEE, 2021.

[13] Denis Foley and John Danskin. Ultra-performance pascal gpu and nvlink interconnect. *IEEE Micro*, 37(2):7–17, 2017.

[14] Georgi Gerganov and ggml-org contributors. llama.cpp: Llm inference in c/c++. https://github.com/ggml-org/llama.cpp, 2025. Accessed June 6, 2025.

[15] ggml org. Issue #8798: [title of the issue]. https://github.com/ggml-org/llama.cpp/issues/8798, 2025. Accessed June 6, 2025.

[16] Hector Marco Gisbert and Ismael Ripoll. Exploiting linux and pax aslr's weaknesses on 32-bit and 64-bit systems. 2016.

[17] Yanan Guo, Zhenkai Zhang, and Jun Yang. GPU memory exploitation for fun and profit. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 4033–4050, Philadelphia, PA, August 2024. USENIX Association.

[18] Jarrett Hoover. Analysis of gpu memory vulnerabilities. 2022.

[19] Daehee Jang. Badaslr: Exceptional cases of aslr aiding exploitation. In Hyoungshick Kim, editor, *Information Security Applications*, pages 278–289, Cham, 2021. Springer International Publishing.

[20] Zhe Jia, Marco Maggioni, Jeffrey Smith, and Daniele Paolo Scarpazza. Dissecting the nvidia turing t4 gpu via microbenchmarking. *arXiv preprint arXiv:1903.07486*, 2019.

[21] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P. Scarpazza. Dissecting the nvidia volta gpu architecture via microbenchmarking. *arXiv preprint arXiv:1804.06826*, 2018.

[22] kokol16. Gpu illegal memory access in kernel densetocsrsparsematrix. https://github.com/tensorflow/tensorflow/issues/92413, 2025. Accessed: 2025-05-30.

[23] Jakob Koschel, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Tagbleed: Breaking kaslr on the isolated kernel address space using tagged tlbs. pages 309–321, 09 2020.

[24] S. Kullback and R. A. Leibler. On information and sufficiency. *The Annals of Mathematical Statistics*, 22(1):79–86, 1951.

[25] Kuterd. nv isa solver. https://kuterdinel.com/nv_isa_sm89/CALL.html, 2024. Accessed December 24, 2024.

[26] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R Tallent, and Kevin J Barker. Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect. *IEEE Transactions on Parallel and Distributed Systems*, 31(1):94–110, 2019.

[27] Hector Marco-Gisbert and Ismael Ripoll. Address space layout randomization next generation. *Applied Sciences*, 9:2928, 07 2019.

[28] Andrea Miele. Buffer overflow vulnerabilities in cuda: A preliminary analysis. *Journal of Computer Virology and Hacking Techniques*, 12(2):113–120, 2016.

[29] Sparsh Mittal, S. B. Abhinaya, Manish Reddy, and Irfan Ali. A survey of techniques for improving security of gpus. *Journal of Hardware and Systems Security*, 2(3):266–285, 2018.

[30] Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael Abu-Ghazaleh. Rendered insecure: Gpu side channel attacks are practical. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 2139–2153, New York, NY, USA, 2018. Association for Computing Machinery.

[31] Ajay Nayak, B. Pratheek, Vinod Ganapathy, and Arkaprava Basu. (mis)managed: A novel tlb-based covert channel on gpus. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security (ASIA CCS '21)*, page 14. ACM, 2021.

[32] Illya Nemenman, Fariel Shafee, and William Bialek. Entropy and inference, revisited. *Advances in Neural Information Processing Systems*, 14:471–478, 2002.

[33] Ruslan Nikolaev, Hassan Nadeem, Cathlyn Stone, and Binoy Ravindran. Adelie: continuous address space layout re-randomization for linux drivers. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 483–498, 2022.

[34] NVIDIA. Cuda c programming guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#device-memory-accesses, 2023. Accessed December 25, 2024.

[35] NVIDIA Corporation. CVE-2020-5991: NVIDIA NVJPEG library Out-of-Bounds READ/Write Vulnerability. https://nvidia.custhelp.com/app/answers/detail/a_id/5094, October 2020. Published October, 2020. Accessed June 6, 2025.

[36] NVIDIA Corporation. Chapter 42. GSP Firmware – NVIDIA Driver Documentation (Version 510.39.01), 2022. Accessed May 30, 2025.

[37] 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, 2020.

[38] Can Peng, Chenlin Huang, Daokun Hu, Di Bang, Jianhua Sun, Hao Chen, and Xionghu Zhong. Address randomization for dynamic memory allocators on the gpu. In *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 570–577. IEEE, 2019.

[39] Sandeep Gogineni Ravindrababu, Varsha Venugopal, and Jim Alves-Foss. Analysis of firmware security mechanisms. In *Intelligent Sustainable Systems: Selected Papers of WorldS4 2021, Volume 1*, pages 537–545. Springer, 2022.

[40] Don Rickman. Powershell script to restart ollama on crash, close or frozen. https://www.edgeventures.com/kb/post/2024/09/04/powershell-script-to-restart-ollama-on-crash-close-or-frozen, September 2024. Accessed June 6, 2025.

[41] Jonas Roels, Adriaan Jacobs, and Stijn Volckaert. Cuda, woulda, shoulda: Returning exploits in a sass-y world. In *Proceedings of the 18th European Workshop on Systems Security*, EuroSec'25, page 40–48, New York, NY, USA, 2025. Association for Computing Machinery.

[42] Gilad Shainer, Ali Ayoub, Pak Lui, Tong Liu, Michael Kagan, Christian R Trott, Greg Scantlen, and Paul S Crozier. The development of mellanox/nvidia gpudirect over infiniband—a new model for gpu to gpu communications. *Computer Science-Research and Development*, 26:267–273, 2011.

[43] Claude E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, 1948.

[44] Tyler Sorensen and Heidy Khlaaf. Leftoverlocals: Listening to llm responses through leaked gpu local memory. *arXiv preprint arXiv:2401.16603*, 2024.

[45] Brad Spengler. Pax: The guaranteed end of arbitrary code execution. *G-Con2: Mexico City, Mexico*, 2003.

[46] Wikipedia contributors. Address space layout randomization, 2025. Accessed December 23, 2024.

[47] yifuwang. Illegal memory access resulted from pointwise autotuning of a cat-like kernel. https://github.com/pytorch/pytorch/issues/127652, 2024. Accessed May 30, 2025.

[48] Chaochao Zhang and Rui Hou. Lak: A low-overhead lock-and-key based schema for gpu memory safety. In *2022 IEEE 40th International Conference on Computer Design (ICCD)*, pages 705–713. IEEE, 2022.

[49] 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 *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, 2023.

Appendix

More Reverse-engineering Details of FlagProbe

According to previous work [17], NVIDIA GPUs use two different address-translation mechanisms. This causes the addresses printed by `cuda-gdb` to differ from the actual addresses recorded in the page tables. Therefore, we cannot trust `cuda-gdb`'s addresses directly; instead, we must locate segment addresses by searching for flags in GPU physical memory and by consulting the page tables.

System constant memory: Raw values read from registers (e.g., `0xfffb88`) produced many collisions. To reduce these, we leverage the alignment property of constant-memory loads: valid flags are 8-byte aligned. We filter out occurrences that are not aligned accordingly. We then corroborate multiple candidate flags by checking multiple constant loads. For example, we search for values produced by `MOV R11, c[0x0][0x118]` and verify that they maintain an expected relative offset (`0x118-0x28=0xF0`). The combination of alignment filtering and relative-offset checks reliably confirms the location of the segment.

.text: We initially attempted to use ordinary GPU instruction sequences as flags, but these conflicted heavily with instructions from GPU CUDA libraries (unknown at first). To avoid such collisions, we try many instruction patterns and finally choose instructions that load from system constant memory (e.g., `MOV R1, c[0x0][0x28]`). Such patterns almost do not appear in CUDA library code and thus can serve as reliable flags. However, after completing the reverse engineering, we found that the text segment addresses printed in `cuda-gdb` were consistent with those we recovered. Nevertheless, `cuda-gdb` failed to display (using `p /x` instruction) the correct contents corresponding to these text addresses and always print `0x0`.

CUDA library: Attempts to obtain CUDA library addresses via `cuda-gdb` or by printing function addresses failed. The `cuda-gdb` would reports missing symbols, and compiling code that prints function addresses did not succeed. We found that NVIDIA treats certain library functions specially: some functions (e.g., `memcpy`) may be inlined into the text segment, while others (e.g., `printf`) are not exposed to symbol-based single-step debugging (e.g., `si` or `ni`) and therefore cannot be stepped into. We observed that branch targets in `.text` disassembly often show as 0x0 until they are resolved at runtime. So by reading branch instruction values from physical memory and resolving their runtime targets, we can identify addresses that belong to the CUDA library.

CPU and CPU Objects Correlation

Figure 6 shows the correlation between CPU-side objects. Most CPU objects exhibit correlation entropy above 25 bits. However, there are still three abnormal situations: the correlation entropy between environment variables and the stack is only about 10 bits, a finding consistent with the results reported in the paper [5]. We additionally observe that the CUDA library, global variables, and the `.text` segment on the CPU share the same ASLR offset block. Moreover, the correlation entropy between the `glibc` address and shared memory (obtained via `shmget/shmat`) is only 12 bits. Consequently, once a GPU-derived leak reveals a CPU-side glibc address, it may enable further disclosure of addresses within the shared-memory region.

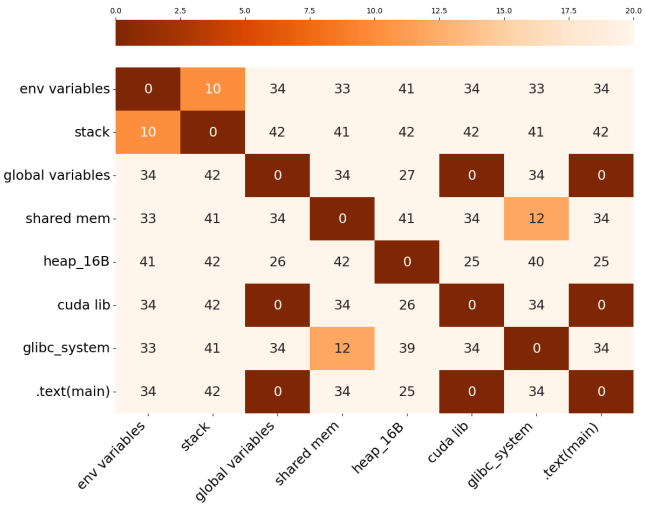


Figure 6: The correlation entropy between different CPU objects. Both the X-axis and Y-axis represent CPU objects.

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 of Paper

This paper presents the first comprehensive study of ASLR on NVIDIA GPUs, a security mechanism well explored on CPUs but largely overlooked in the GPU domain. By introducing new methods to probe GPU memory mappings, the authors reconstruct detailed layouts and assess ASLR strength using entropy-based metrics.

The analysis uncovers critical weaknesses: GPU heaps are left unrandomized, and consistent offset patterns link GPU and CPU memory. These flaws significantly undermine the protection offered by ASLR, leaving both GPU and host CPU memory layouts more vulnerable than expected. The authors validate their findings with NVIDIA and demonstrate a practical attack that leverages GPU information to infer CPU memory organization. Beyond identifying these risks, the work highlights the importance of reverse engineering closed GPU architectures as a foundation for both exposing vulnerabilities and strengthening future defenses.

A.2. Scientific Contributions

- Independent Confirmation of Important Results with Limited Prior Research
- Identifies an Impactful Vulnerability
- Creates a New Tool to Enable Future Science

A.3. Reasons for Acceptance

1) The paper delivers the first in-depth investigation of ASLR on NVIDIA GPUs. As the study demonstrates, many earlier assumptions and preliminary findings do not hold, making this work highly valuable for researchers and practitioners focused on GPU-based attacks and defenses.
2) The authors created and released a framework with novel techniques for analyzing GPU memory layout, offering an important resource for the community.
3) All vulnerabilities were disclosed responsibly to the vendor, and any potentially sensitive information was carefully anonymized to prevent misuse.