Mitigating Code Injection Attacks on Modern GPUs

Carlo Ramponi
University of Trento
Italy

Adriaan Jacobs
DistriNet, KU Leuven
Belgium

Luigi Dell'Eva
University of Trento
Italy

Stijn Volckaert
DistriNet, KU Leuven
Belgium

Silviu Vlasceanu
Huawei Research
Germany

Mahmoud Ammar
Huawei Research
Germany

Bruno Crispo
University of Trento
Italy

Abstract

Once specialized for graphics acceleration, Graphics Processing Units (GPUs) now serve as foundational components of parallel computing, driving advancements from immersive gaming to generative Artificial Intelligence (AI). However, while their computational capabilities have expanded dramatically, GPU security has lagged behind their expanding functionality. Recent studies reveal that GPUs remain susceptible to memory corruption attacks, a gap amplified by their reliance on memory-unsafe programming frameworks, such as CUDA. Although contemporary GPUs incorporate basic defenses like ASLR and stack canaries, the absence of hardware-enforced Data Execution Prevention (DEP) leaves them critically exposed to code-injection attacks, undermining the implementation of a robust W⊕X security policy.

This work analyzes the potential of code-injection attacks on NVIDIA GPUs, executed independently of CPU vulnerabilities, in the presence of ASLR and stack canary protections. To address this gap, we design and implement a software-based DEP mechanism, a critical enabler of the W⊕X policy, to enforce strict memory permissions, eliminating executable-writable memory regions, and thereby neutralizing code-injection attacks. Our solution operates transparently alongside proprietary vendor toolchains and software components while maintaining compatibility with precompiled libraries. Empirical evaluations across diverse benchmarks on three different NVIDIA GPU architectures show that our approach incurs minimal overhead, with average runtime and memory overheads of less than **0.5%**.

Extended author version. This manuscript is the extended version of a paper accepted for publication in the *Proceedings of ACM CCS 2026*. It includes an appendix that was part of the peer-reviewed submission but was omitted from the camera-ready proceedings version due to space constraints. DOI forthcoming.

1 Introduction

Graphics Processing Units (GPUs) have evolved from specialized graphics hardware into powerful, general-purpose parallel computing engines [15]. They now underpin workloads ranging from large-scale machine learning to generative AI, with recent systems such as DeepSeek illustrating this transformation [33]. GPU vendors continuously advance architectures for both datacenter and consumer scenarios. However, despite this rapid expansion in capability, GPU security features have not matured at the same pace [11, 23, 67]. This divergence between adoption and security hardening is particularly concerning given the increasingly important role of GPUs in many fast-growing industries [28, 74, 84]. For instance, OpenAI leveraged NVIDIA's Volta V100 GPUs to train GPT-3 [13]. Similarly, DeepSeek's recent models at the time of writing, including DeepSeek-V3 and DeepSeek-R1, were reported to rely primarily on large-scale NVIDIA GPU clusters, including Ampere A100 and Hopper H100 GPUs [33].

Problem Statement. As a result, GPUs have become attractive targets for a range of attack vectors [18, 21, 23, 32, 61, 67, 68, 74, 80], with memory corruption attacks among the most significant, as confirmed by vendors such as NVIDIA [71, 76]. For instance, recent research has shown that memory corruption on GPUs can be exploited to leak confidential information [70], alter computation outcomes [61], hijack resources [74], or compromise the host system [88]. Security researchers have demonstrated techniques akin to classic CPU exploitation, including code injection and forms of code reuse attacks like Return-Oriented Programming (ROP) [23, 61, 67]. In response, researchers have begun proposing GPU-specific sanitizers and testing tools to detect memory safety violations and mitigate exploitation in GPU applications [28, 29, 71, 76].

These approaches improve the detection of memory safety violations, but leave a key exploitation barrier unaddressed. Once memory corruption occurs, current GPU platforms do not reliably prevent writable, attacker-controlled memory from being executed as code. Consequently, an undetected memory safety violation may still be transformed into a code-injection attack, revealing a fundamental gap between vulnerability detection and exploit mitigation.

This gap stems from missing or only partially realized hardware primitives for instruction-data separation on GPUs. Basic defenses that have been standard on CPUs for decades, most notably hardware-assisted Data Execution Prevention (DEP) [34], are absent or inconsistently implemented in current GPU architectures [23, 67], as shown in Table 1. This is particularly risky given that general-purpose GPU programming frameworks like CUDA [47] and OpenCL [39] extend memory-unsafe languages such as C and C++, and therefore inherit decades-old vulnerabilities endemic to these languages [76]. Although recent GPU designs incorporate mitigations such as stack canaries and address space

layout randomization (ASLR) [67], they still do not provide the combination of non-executable data pages and non-writable code pages needed to enforce a robust W⊕X policy. In CPUs, W⊕X is enforced by the Memory Management Unit (MMU) through two hardware-level page table flags: one marks data memory as non-executable, enabling DEP, and another marks code memory as non-writable. In contrast, starting with Volta, recent NVIDIA GPU architectures expose only a single GPU MMU (GMMU) flag for marking code pages as non-writable (see Table 1). This feature is disabled by default and remains challenging to configure. Even if this protection is enabled, the absence of DEP means that attacker-controlled data may still be executed as code, leaving code injection viable in the presence of memory corruption.

Contributions. This paper examines the feasibility of code injection on recent NVIDIA GPUs, under a threat model where ASLR and stack canaries are enabled. Our findings show that the lack of effective W⊕X enforcement leaves code injection as a potential attack vector in vulnerable CUDA programs on current GPU platforms. However, practical exploitability depends on GPU architectural constraints, the memory-corruption primitive exposed by the vulnerable program, and the degree of adversarial control over its inputs and execution state. We further analyze the challenges of configuring and reliably enforcing non-writable code pages, despite the presence of a dedicated GMMU permission bit for this purpose. While W⊕X is mature and well understood on CPUs [6, 9, 26, 34, 69], our study shows that it is neither reliably supported nor straightforward to deploy on GPUs today.

We then design and implement `cuDEP`, a lightweight software-assisted DEP mechanism for GPUs that prevents execution from data pages. Building on this DEP foundation, we enforce a practical W⊕X policy whose second component uses transparent runtime interposition to set the GPU code-page permission bit and deny writes to code pages during execution. Our evaluation shows that effective W⊕X enforcement is feasible on current GPUs despite proprietary hardware and software constraints. Experiments across several benchmarks on three NVIDIA GPU architectures, namely Ampere, Hopper, and Blackwell, demonstrate that our approach incurs less than **0.5%** average runtime and code-size overhead while remaining compatible with pre-compiled libraries. Finally, our results provide an immediate mitigation for existing GPU deployments and a clear call for accelerator vendors to introduce first-class hardware support and safer defaults in future architectures.

Note. The source code for our solution and the accompanying evaluation artifacts are publicly available [12].

2 Background

2.1 Overview of NVIDIA GPU Architecture

Figure 1 depicts a generic NVIDIA GPU architecture [57]. At the core are *Streaming Multiprocessors* (SMs), each of which contains multiple simple cores, load/store (LD/ST) units, special function units (SFUs), and a large register file. The SM scheduler groups threads into *warps*, typically of size 32, and executes them in a *Single-Instruction Multiple-Thread* (SIMT) fashion, allowing thousands of threads to run simultaneously. Each SM includes a private L1 cache and an on-chip shared memory, while all SMs connect through an on-chip interconnect to a device-wide L2 cache. The L2 cache

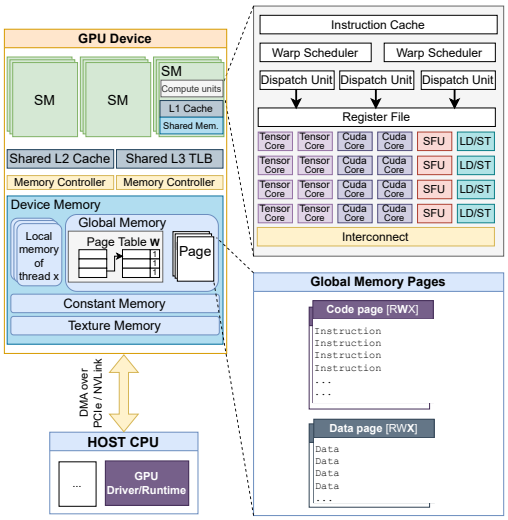

Figure 1: Overview of a generic NVIDIA GPU architecture.

interfaces with multiple memory controllers, which provide access to high-bandwidth off-chip device memory (e.g., GDDR6 or HBM2). Furthermore, each SM fetches instructions from its instruction cache and dispatches them to the relevant computing units, with threads executing synchronously within a warp. NVIDIA GPUs provide a large register file, with tens of general-purpose registers per thread (e.g., 256 in Ampere), alongside predicate and other special-purpose registers [17].

GPU programming models such as CUDA expose multiple memory spaces with distinct roles. *Global memory* resides in off-chip DRAM and serves as the primary heap for program code and data; it is accessible to all threads within a GPU application and cached at the L1 and L2 levels. *Local memory* is a per-thread region used mainly as a stack for spilled registers and private data; despite its name, it is allocated in device DRAM and remains private to the owning thread. *Shared memory* is an on-chip scratchpad shared by threads within a block, enabling low-latency data sharing and synchronization. *Constant* and *texture memory* are read-only spaces optimized for broadcast and 2D/3D spatial locality, respectively.

Although the hardware distinguishes these memory spaces, they are unified under a single virtual memory system. Each space is accessed via dedicated instructions (e.g., `LDG/STG` for global, `LDL/STL` for local, and `LDS/STS` for shared memory), while generic load and store instructions (`LD/ST`) operate across all memory spaces [47].

2.2 Virtual Memory on NVIDIA GPUs

Modern GPUs implement virtual memory to provide isolation across GPU programs (i.e., GPU contexts). Each GPU program maintains its own multi-level page table in the global memory, managed by the GPU driver. The GMMU handles address translation for memory requests issued by SMs [50] by walking a page-table to locate the corresponding page table entry (PTE), caching intermediate results in translation lookaside buffers (TLBs) at both the SM level (L1 TLB) and the device level (L3 TLB) [81].

Table 1: Comparison of Features Across Recent and Older NVIDIA GPU Architectures [42]. ✗ indicates the feature is unavailable, ✓ means the feature is available and actively used, ✓ indicates the feature is available but not utilized, and ? represents unknown status. All data is based on experimental evaluations or published reports [23, 35, 45, 56, 61, 67].

Architecture	Year	Target	Big New Thing	Consumer Examples	Datacenter Examples	W+X Policy	
						NX Bit (DEP)	Bit for Non-Writable Code
Blackwell	2024	Datacenter	Multi-die (chiplet) GPUs, FP4 precision	N/A	B100, B200	✗	✓
Hopper	2022	Datacenter	Transformer Engine, FP8	N/A	H100	✗	✓
Ada Lovelace	2022	Consumer	DLSS 3 (AI frame generation), 3rd-gen RT cores	RTX 4090, 4080, 4070	N/A	✗	✓
Ampere	2020	Both	Structural sparsity, 2nd-gen RT and Tensor Cores	RTX 3090, 3080, 3070	A100, A30	✗	✓
Turing	2018	Consumer	1st real-time ray tracing (RTX), Tensor Cores for DLSS	RTX 2080, 2070, 2060	T4 (for inference)	✗	✓
Volta	2017	Datacenter	Tensor Cores introduced (AI acceleration)	N/A	V100	✗	✓
Pascal	2016	Both	NVLink, big perf/watt gains, early HBM2	GTX 1080, 1070, 1060	P100	✓	✓
Maxwell	2014	Consumer	Major efficiency boost, CUDA improvements	GTX 980, 970	N/A (some Tesla M40)	✓	✗
Kepler	2012	Both	Dynamic Parallelism, Hyper-Q	GTX 680, 670	Tesla K80	✓	?

Recent NVIDIA GPUs employ a 49-bit virtual address space with five levels of translation. PTEs encode physical frame mappings as well as access permissions. However, according to the page table format released by NVIDIA [47, 56], validated by our experimental analysis and prior work [23], the GMMU lacks an execute-permission bit to differentiate code from data, meaning that DEP is not supported. Moreover, although the format specifies a non-writable bit for code pages, this bit is never set in practice, leaving code pages writable. A summary of the availability of these bits across different NVIDIA architectures is provided in Table 1.

2.3 Host–Device Interaction

CUDA adopts a heterogeneous programming model in which the GPU acts as a coprocessor to the CPU. Applications are typically written in C or C++ and consist of *host code*, which executes on the CPU, and *device code*, which executes on the GPU. Device code comprises *global functions* (aka CUDA *kernels*) invoked from the host and executed on the GPU, and *device functions*, which can only be called from kernels or other device functions. Host code interacts with the GPU via the CUDA runtime, which manages memory allocation, kernel launches, and context handling. CUDA exposes a high-level *Runtime API*, which offer functions like `cudaMalloc` and `cudaLaunchKernel`, as well as a lower-level *Driver API* that provides explicit control over contexts and module loading. The two APIs can be used independently or combined [37, 38]. The CUDA runtime is distributed as part of the CUDA Toolkit, which also includes the `NVCC` compiler, core libraries (e.g., `libcudart`, `cuBLAS`), and development tools such as `CUDA-GDB` [54].

The compilation of CUDA programs is orchestrated by `NVCC`, which preprocesses source files and separates host and device code. The host code is further compiled by a standard compiler (e.g., `GCC` or `Clang`) into CPU object code, while the device code is translated into `Parallel Thread Execution (PTX)`, an intermediate representation that is further compiled into fixed-length 16-byte native GPU instructions (`Streaming Assembly (SASS)`). This translation is performed by the `PTX` assembler (`PTXas`), which operates either ahead-of-time (during compilation) or just-in-time (JIT) under the control of the CUDA runtime [8, 55]. The final output is a standard ELF executable containing a special `FATBIN` (fat binary) section, which encapsulates portable `PTX` and/or architecture-specific `SASS` (`cuBIN`) binaries [24, 53]. At program launch, the CUDA runtime extracts the FATBIN, initializes a CUDA context, performs JIT compilation of `PTX` if needed, allocates GPU memory pages for code and data (with default read, write, and execute permissions), and orchestrates kernel execution. When a kernel is launched, threads are organized into a grid of blocks that are scheduled across available SMs, while host–device data transfers are handled via DMA engines over high-speed interconnects such as PCIe or NVLink [36].

3 Code Injection on Modern GPUs

Recent research [23] demonstrated shellcode injection on NVIDIA's vGPU platform in the absence of mitigations such as ASLR and stack canaries. In contrast, this section characterizes code injection against vulnerable CUDA programs running on modern GPUs with these mitigations enabled, identifying the residual attack surface and constraints that persist under this threat model. For clarity, this section and the next use the Ampere architecture as a running example, while Section 5 discusses the extent to which our findings generalize to other GPU architectures.

Scope. Our analysis is intentionally conducted at the level of controlled, vulnerable CUDA programs. Our objective is not to establish end-to-end exploitability against deployed ML applications or serving infrastructures. Instead, we isolate and characterize the barriers imposed by ASLR and compiler-inserted stack canaries on modern NVIDIA GPUs, identify the capabilities required to bypass them, and determine whether code injection remains possible once those capabilities are available.

3.1 Threat Model

We consider an adversary targeting a vulnerable CUDA program executing on a modern NVIDIA GPU. The host application, CUDA runtime, and driver are assumed to be uncompromised. Rather than modeling how malicious inputs propagate through an application or serving stack, we consider attacker-controlled values that directly reach the vulnerable device code. In particular, we assume:

- The GPU executes with all currently available vendor-provided defenses enabled, including ASLR and stack canaries.
- The adversary has identified two memory safety errors in device code: (i) an out-of-bounds (OoB) read capable of disclosing runtime values, including randomized addresses and stack canaries, and (ii) an OoB write capable of corrupting control data, such as saved return addresses in local memory or function pointers in global memory.
- The adversary can control the values supplied to the vulnerable program and observe its externally visible output or failure. For analyses involving repeated guesses, we explicitly distinguish attempts that can occur within the same CUDA context from those requiring a host-process restart.

These assumptions deliberately isolate the effectiveness of ASLR and stack canaries from vulnerability discovery and input reachability. We do not claim that commodity CUDA applications, ML frameworks, or serving systems expose this combination of primitives, or that remote users can exercise them with the assumed degree of control. Establishing end-to-end exploitability in deployed ML inference stacks remains challenging, although recent work has taken a step in this direction by demonstrating code injection through memory errors in real-world GPU software under more realistic application-level conditions [66].

3.2 GPU-specific Adversarial Knowledge

This section examines whether ASLR and stack canaries introduce GPU-specific challenges distinct from those observed on CPUs [65].

3.2.1 ASLR. To evaluate the effectiveness of ASLR, we first sought to understand the semantics and exact ranges of GPU virtual memory segments for CUDA applications when address randomization is disabled. To this end, we developed several sample CUDA applications, compiled them with `-O0` optimization flag, and executed them on our test platform. The choice of `-O0` optimization allows us to print virtual addresses of variables residing in different GPU memory spaces, including global, local, and shared memory, using `printf` statements inside `device functions`. This is not possible under higher optimization levels (e.g., `-O3`) due to aggressive inlining of device functions (i.e., a constant value of `0x8` is printed).

We further leveraged a tool released by Zhenkai et al. [85] to extract the GPU page tables for each evaluated application. By correlating the printed virtual addresses with the retrieved address ranges, we inferred the relevant structure of the page table layout. Given this fixed layout, we can identify the address ranges corresponding to different memory spaces irrespective of whether ASLR is enabled. We then enabled ASLR and repeated the same procedure for the selected applications. Table 7 in Appendix A reports the observed start addresses of global, local, shared, and user-defined constant memory across multiple runs of a representative application on Ampere GPUs with ASLR enabled. The table also includes the base address of the GPU heap allocated from host code via `cudaMalloc`, which resides in global memory, as well as the device heap used for dynamic allocations issued from within device code.

Findings. ① We found that ASLR computes the randomized base addresses of all GPU memory regions **at the launch of the CPU host process** (i.e., initialization of a CUDA context), except for addresses of dynamic allocations from within device code, which remain unrandomized. However, we could not find any public evidence of the usage of device heap allocation in open-source software, including our selected benchmarks in Section 7, suggesting that this might **not** be a critical attack vector. ② We also noticed that these randomized base addresses **remain fixed** for the process's lifetime and are stored in read-only constant memory, from which they are retrieved for every subsequent kernel launch within the same process. ③ On our **Ampere GPU**, which employs a 49-bit virtual address space, **we found that only 20 bits of the virtual address space are randomized, specifically bits [44:25] for all memory regions**. For instance, on our platform, global memory consistently begins at `0x7[20 random bits]400100`, whereas local memory always starts at `0x7[random 20 bits]fffb68`. This observation of limited entropy aligns with findings on the older Turing architecture [67] as well as a recent work on the Ada architecture [87], suggesting that GPU ASLR is more susceptible to leakage or brute-force attacks than its CPU equivalent. ④ Furthermore, by collecting the starting addresses of multiple GPU memory spaces across 40,000 executions of diverse CUDA applications on our Ampere platform, we observed two consistent patterns. First, for certain pairs of memory spaces, **the offset between their base addresses remains fixed**. Specifically, the offset is 16 bits between global and constant memories (0x100) and 32 bits between local and shared memories (0x01000498). Second, **other memory spaces exhibit a limited range of 32-bit offset values in which variation is confined to the most significant byte**, implying at most 2^8 distinct offsets. Table 6 in Appendix A summarizes the observed offset ranges across several memory spaces. The prevalence of such relative addressing in ASLR-enabled GPU applications highlights an additional weakness compared to CPU-based systems: a single leaked randomized base address could facilitate further leaks, exposing a large part of the memory layout.

Brute-forcing ASLR in Black-box Settings. To evaluate brute-forcing in black-box settings, we attempted to infer randomized addresses in applications compiled with `O3`, where no address disclosures are available. In practice, we found that naïvely probing random addresses is ineffective due to the error handling semantics of CUDA [31, 54]. Accessing an unmapped or invalid address from device code triggers an illegal memory access, which results in a **sticky error** that corrupts the entire CUDA context [19, 82]. Once such an error occurs, all subsequent CUDA API calls fail, and the only recovery mechanism is to terminate and relaunch the host process, which in turn re-randomizes all virtual addresses. This behavior fundamentally limits straightforward brute-force attacks, as each failed guess collapses the probing session.

Nevertheless, our experiments identify narrow conditions under which brute-forcing may still succeed despite sticky errors. If the adversary can confine guesses to a known mapped address range, probing avoids unmapped pages and incorrect attempts trigger only non-sticky errors [31, 82], which do not require host process relaunch. Additionally, in fork-based execution models, ASLR randomization occurs once at the parent process launch and is inherited by all children. A child process may therefore trigger a sticky error and terminate without corrupting the parent's CUDA context, enabling repeated brute-force attempts. While such patterns are uncommon in CUDA applications and can be mitigated through proper error handling, they demonstrate that brute-force resistance is not absolute. We conclude that successful brute-forcing requires (i) knowledge of the target memory space (e.g., global or local), (ii) an approximate allocation range inferred from the fixed GPU page table layout, and (iii) a probing primitive that avoids immediate process termination. In contrast to prior claims that GPU ASLR is broadly vulnerable to brute-force attacks [67], our findings show that the CUDA sticky error mechanism [19, 31, 82] substantially increases the cost of brute-forcing GPU ASLR by enforcing process-wide termination upon invalid memory accesses.

3.2.2 Stack Canaries. Stack canaries are disabled by default in `NVCC` and can be enabled via `-device-stack-protector=true`. Comparing SASS code compiled with and without stack protection reveals

a consistent prologue–epilogue pattern in which a canary is loaded from constant memory, stored on the stack, and verified before control returns to the caller.

Findings. On Ampere GPUs, ① we observed that the **canary consists of two 4-byte values**, with the least significant byte fixed to `0x00`, mirroring CPU conventions. As shown in Figure 2a, the canary is loaded from fixed constant-memory locations (`c[0x0][0x150]` and `c[0x0][0x154]`) and stored via `STL.64`. The epilogue (Figure 2b) reloads the canary and compares it against the stack value using two `ISETP.EQ.U32.AND` instructions, each corresponding to one half of the canary. Execution proceeds on a match and aborts via `BPT.TRAP` otherwise. For clarity, Figure 2 shows the SASS sequence emitted under `-O0`. While higher optimization levels may alter the instruction sequence, the canary location and the load–store–check logic remain unchanged.

```
MOV R6, c[0x0][0x150] ;
MOV R7, c[0x0][0x154] ;
STL.64 [R1+0x410], R6 ;
```

(a) Push instructions

```
MOV R4, c[0x0][0x150] ;
MOV R5, c[0x0][0x154] ;
LDL.64 R6, [R1+0x410] ;
ISETP.EQ.U32.AND P0, PT, R4, R6,
     PT ;
ISETP.EQ.U32.AND P0, PT, R7, R5,
     P0 ;
@P0 BRA 0xc80 ;
BPT.TRAP 0x1 ;
/*0c80*/
```

(b) Verification instructions

Figure 2: SASS instructions for stack canary handling.

② We also observed that stack canaries, like randomized addresses, **are not regenerated** with each CUDA context launch from the same host process, and that the **canary value remains fixed** across `global` and `device functions` within the same context, consistent with prior findings on the Turing GPU architecture [67]. We further analyzed the heuristics used by NVCC to determine when stack canaries are inserted in `device functions` on the Ampere architecture. Using test cases compiled with `-O3` to evaluate optimized CUDA binaries, ③ we found that **canary insertion is governed by a combination of inlining decisions, static control-flow analysis, and stack-frame size**. First, although aggressive inlining of `device functions` substantially reduces canary coverage, compiling with relocatable device code (`-dc`) preserves function boundaries and enables canary insertion. Second, canary insertion depends on the compiler's ability to statically reason about loop bounds. Canaries are omitted when all array accesses are provably in bounds at compile time, but inserted when potential OoB accesses are detected, except when external library calls are present, in which case canaries are omitted regardless. Finally, functions with stack frames of at most 40 bytes never receive stack canaries, even when return addresses are spilled to the stack. This is reflected in the function prologue, where the `IADD3 R1, R1, <offset>, RZ` instruction adjusts the stack pointer, and offsets of $\leq$ `-0x40` result in no canary insertion.

Brute-forcing Stack Canaries. Similar to ASLR, an incorrect guess during stack-canary brute-forcing triggers a kernel failure that results in a **sticky error** [19, 82], corrupting the CUDA context and forcing termination of the host process. While fork-based execution can, in limited cases, amortize sticky errors when brute-forcing ASLR, we found that this strategy does not apply to stack canaries. Unlike common CPU operating systems, where forked processes inherit the same canary value, each CUDA context is initialized with a fresh stack canary, yielding different values across parent and child processes. Consequently, contrary to prior claims [67], repeated guessing across executions does not accumulate progress, rendering brute-forcing stack canaries infeasible in practice and leaving information leakage as the primary viable avenue.

3.3 Controlled Demonstration of Code Injection

To determine whether ASLR and stack canaries constitute complete barriers to code injection, we construct two proof-of-concept CUDA programs satisfying the assumptions in Section 3.1. Building on the findings in Section 3.2, Figure 3 illustrates the two attack variants demonstrated on Ampere GPUs with both mitigations enabled. Both variants use information disclosure, rather than brute force, to recover the runtime values required for exploitation.

The first variant (top path in Figure 3) injects shellcode into data memory. An OoB read discloses the stack canary and randomized GPU memory addresses, which are used to construct the SASS payload. An OoB write then overwrites a spilled return address with the payload address while restoring the leaked canary. Upon function return, control is redirected to the payload, which executes directly from data memory.

The second variant (bottom path in Figure 3) exploits writable GPU code pages. After disclosing the address of the target code region, an OoB write replaces existing SASS instructions with attacker-controlled instructions. These instructions execute when the program counter reaches the modified region, without requiring a separate control-flow redirection.

Further implementation details and reproduction instructions are available in our accompanying artifact [12].

4 W⊕X Policy for Modern NVIDIA GPUs

Overview. Our objective is to enforce a practical W⊕X policy on NVIDIA GPUs within the existing CUDA software stack. The novelty of this design does not lie in the W⊕X principle itself, which is well established in CPU systems, but in realizing this principle on a platform where the usual CPU assumptions do not hold. On CPUs, W⊕X enforcement typically relies on architectural execute-permission bits, documented page-table interfaces, and operating-system control over code loading and permission transitions. In contrast, current NVIDIA GPUs do not expose a Never-eXecute (NX) bit for distinguishing executable and non-executable pages, and GPU code pages are allocated, populated, and mapped internally by the proprietary CUDA runtime and driver. As a result, application code cannot simply request conventional code and data permissions through a documented API.

Realizing W⊕X on NVIDIA GPUs therefore requires two complementary mechanisms. First, GPU code pages must be reliably transitioned to a read–execute state by setting the GMMU non-writable bit *after* instructions are transferred into GPU memory and *before* kernel execution begins. Second, because current GMMUs lack an

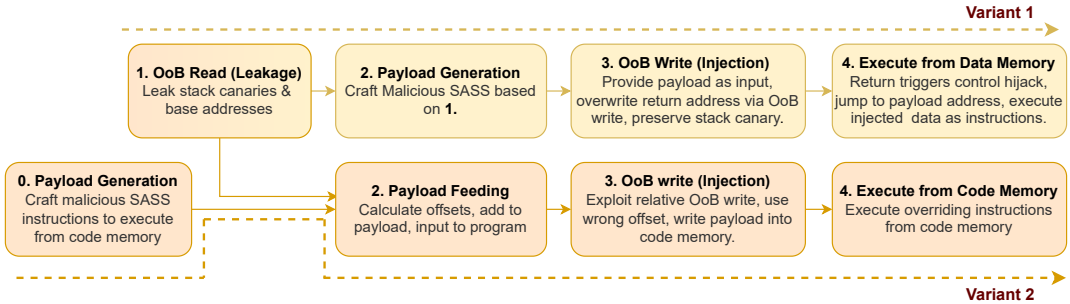

Figure 3: Exploitation flow of two variants of code injection attacks on GPUs, demonstrated despite ASLR and stack canaries.

NX bit, we introduce cuDEP, a lightweight software-based DEP mechanism that prevents the program counter (PC) from reaching data memory by validating unsafe indirect control transfers at the SASS level. This enforcement must happen after ASLR has fixed the runtime code layout, must remain compatible with precompiled and JIT-loaded CUDA binaries, and must cover both the CUDA Runtime and Driver APIs. The rest of this section describes how we obtain the required runtime information, instrument unsafe SASS-level control transfers, and enforce code-page write protection transparently despite the proprietary CUDA ecosystem.

4.1 Challenges

Designing a practical W⊕X policy for target GPUs presents several non-trivial challenges arising from architectural constraints and the proprietary nature of the CUDA software stack.

❶ **Proprietary Ecosystem.** NVIDIA's GPU software stack is largely opaque. Core components of the CUDA toolkit, including the NVCC compiler, CUDA runtime, and the PTX assembler, are closed source. The native SASS ISA is also proprietary and undocumented. Consequently, there is limited public insight into kernel loading, control-flow handling, or safe points for instrumentation.

❷ **Limitations of the PTX Abstraction.** Although PTX is openly documented and has been used by prior security mechanisms, including sanitizers [76] and bounds-checking proposals [62], it is insufficient for enforcing DEP-like defenses. PTX intentionally abstracts away low-level details such as memory layout, segment boundaries, and register allocation. As a result, instrumentation at this intermediate level cannot precisely identify code and data regions or prevent the PC from jumping into data memory. This limitation is exacerbated by CUDA's handling of return addresses: unlike CPUs that use dedicated link registers (e.g., ARM's LR), spilled return addresses are placed in general-purpose registers that vary across functions. Our analysis of multiple cuBIN images using `nvdisasm` [52] confirms this behavior, further complicating return instruction handling at the PTX level.

❸ **Impact of ASLR.** Modern GPUs enable ASLR by default, randomizing code and data base addresses at host process launch. While ASLR raises the bar for adversaries, it complicates W⊕X enforcement, as correctly realizing such a defense requires precise knowledge of code and data boundaries. This challenge is further exacerbated by ❶: the proprietary nature of the GPU software stack obscures the details of ASLR, even though accurate base addresses are essential for both setting the GMMU non-writable bit and enforcing cuDEP execution checks.

❹ **Undocumented FATBIN Format.** GPU binaries are packaged in FATBIN containers that encapsulate PTX and SASS and are parsed exclusively by the proprietary CUDA runtime. The mechanisms by which code pages are allocated and mapped remain undocumented, obscuring both the timing and the location at which permission updates should be applied. Although individual cuBINs are ELF-formatted, they are sometimes compressed, and NVIDIA documentation (e.g., the `nvFatbin` user guide [53]) omits low-level details of the FATBIN format, including header structures, section layouts, and binary encodings.

❺ **Multiple Developer APIs.** As discussed in Section 2.3, NVIDIA exposes two host-side APIs for GPU programming, which may be used independently or in combination. This duality requires defenses like W⊕X to account for multiple kernel-loading and memory-allocation paths to ensure comprehensive protection. Moreover, neither API provides a mechanism to set the GMMU non-writable bit for code pages, leaving it unset by default.

Although CUDA exposes virtual-memory management APIs, such as `cuMemSetAccess`, for controlling access permissions on application-managed GPU memory regions, this control does not extend to device code installed during module loading. CUDA modules are loaded into the current context by the driver, which allocates the resources required for their functions and data; these allocations are exposed through module and function handles rather than as ordinary application-managed virtual-memory regions [58, 59].

4.2 Design Decisions

Our design is guided by the five key challenges identified in the previous section. Challenge ❶ requires that the solution remains **compatible with NVIDIA's proprietary software stack**, as otherwise it would not be deployable in practice. Challenge ❷ dictates that DEP enforcement must **operate at the SASS level**. While this enables instrumentation of closed-source libraries, it is complicated by limited public documentation and the absence of an official SASS assembler, necessitating custom tooling to reassemble modified binaries. Challenge ❸ demands a **post-ASLR design** that correctly applies code-page permissions and injects appropriate operands into instrumentation instructions, particularly those for address-range validation, at load time, once randomized base addresses are known. Challenge ❹ further requires partial reconstruction of the undocumented FATBIN format to **transparently instrument cuBINs during loading**. Finally, Challenge ❺ mandates **support for both the CUDA Runtime and Driver APIs**, as each exposes

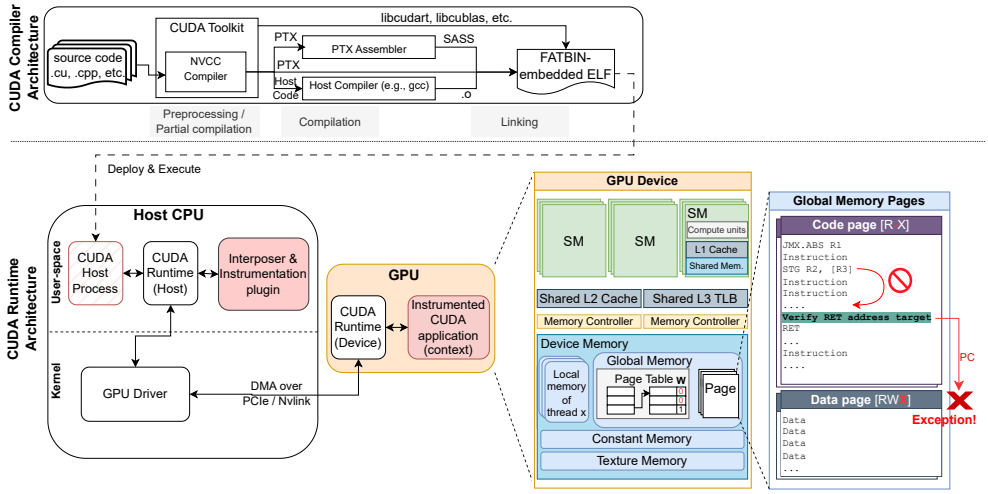

Figure 4: Overview of our solution within the CUDA runtime architecture. The figure shows how W⊕X is enforced by interposing CUDA runtime calls to (i) mark code pages as non-writable after loading and before execution, and (ii) enable cuDEP, a software-based DEP mechanism that validates indirect control transfers to prevent execution from data memory.

distinct entry points for memory allocation and kernel execution, and code pages must be marked as non-writable along both paths.

To realize these characteristics, our design pursues three goals:

- Identify unsafe SASS-level control-transfer instructions that adversaries could exploit in code injection attacks.
- Analyze the CUDA application launch process to determine where ASLR is applied and which system calls must be interposed to extract the required runtime information.
- Develop mechanisms to extract, instrument, and reassemble SASS instructions that integrate seamlessly with proprietary components of the NVIDIA GPU ecosystem.

Figure 4 provides an overview of our solution as realized entirely within the CUDA runtime architecture. Rather than requiring any modifications to the compilation pipeline, we deploy a runtime interposer that hooks the CUDA Runtime and Driver APIs at module load and kernel launch. The interposer extracts the necessary post ASLR layout information, instruments cuBINs on the fly to enforce cuDEP checks, and sets the GMMU non-writable bit for code pages after code is loaded into GPU memory and before execution, effectively enforcing W⊕X as illustrated on the device side in Figure 4. By concentrating all enforcement at runtime, our design becomes largely agnostic to the frontend used to produce GPU binaries, improving portability across stacks that do not necessarily rely on NVCC, such as Clang/LLVM-based CUDA toolchains and ML/DSL compilers (e.g., TVM [14], Triton [77], etc.) that ultimately deploy kernels via the CUDA runtime or driver interfaces.

4.3 Identifying Unsafe Control-Transfers

Building on prior work that demystifies CUDA binaries [8, 24, 27], we performed an extensive analysis of NVIDIA's SASS ISA across several open-source applications (also used in our evaluation in Section 7). We identify four indirect control-transfer instructions susceptible to hijacking: BRX, JMX, CALL R_i, and RET. Each of these instructions may consume operands from registers that can be influenced by an adversary (e.g., return addresses spilled to local memory). We focus the remainder of our design discussion on the RET instruction, as it constitutes the most practical and frequently encountered attack surface [23, 67]. Our analysis further suggests that other indirect branch instructions rarely occur in real-world workloads. To substantiate this observation, we analyzed all `device functions` in NVIDIA's core CUDA libraries. As shown in Table 8 in Appendix B, such indirect branches are extremely rare in these libraries.

```
1 0x0080:  S2R R0, SR_TID.X ;
2 0x0090:  HFMA2.MMA R3, -RZ, RZ, 0, 0 ;
3 0x00a0:  RET.REL.NODEC R2 ;          // Relative return
4 0x0230:  MOV R2, 0x250 ;
5 0x0240:  CALL.REL.NOINC $_Z6kernelPi$_Z4leafv ; // Relative call
6 0x0250:  IADD3 R5, R0, R21, RZ ;
7 0x0260:  IMAD.MOV.U32 R21, RZ, RZ, 0x0 ;
8 0x0270:  RET.ABS R20, 0x0 ;  // Absolute return to address [R21][R20+0x0]
```

Listing 4.1: Sample SASS instructions with relative and absolute return Instructions.

RET Instructions. Return instructions in SASS support both relative and absolute addressing modes [46]. A relative RET specifies an offset within a designated code region (commonly used when functions are packed within the same ELF section), whereas an absolute RET specifies a full virtual address rather than an offset from a base. Listing 4.1 illustrates both modes. In the example, the relative RET (line 3) uses R2 as a 32-bit register holding the offset. By contrast, the absolute RET (line 8) encodes the target address across two registers: R20 stores the lower 32 bits and R21 stores the upper 32 bits. Given the 2MB page size, the upper part is fixed to `0x7fff` on our platform when ASLR is disabled, and the base address is page-aligned. Finally, when sufficient registers are available, return addresses need not be spilled to the stack and may instead reside in general-purpose registers that are later consumed by the RET instruction.

```
...
0x00a0:        BRA `(.L_x_37) ;
...
0x0270:        BRA `(.L_x_38) ;
...
.L_x_37:
 /* Check whether the relative offset is within the bounds of the page */
 0x1080:        ISETP.GE.U32.AND P0, PT, R2, 0x0, PT ;                    // Predicate Register P0 is set (True) if R2 >= 0.
 0x1090:        ISETP.LT.U32.AND P0, P0, R2, 0x2980, PT ;                 // P0 = P0 && (R2 < 0x2980), where 0x2980 is the section size.
 0x10a0: @P0 RET.REL.NODEC R2 `(_Z6kernelPiS_S_) ;                        // Return only if the conditions above hold (target address is valid)
 0x10b0:        BPT.TRAP 0x1 ;                  // Trigger a Trap exception that cause a kernel crash due to ivalid return outside section boundary
.L_x_38:
 /* Check whether the lower part of the absolute return address is within the page bounds */
 0x10c0:        ISETP.GE.U32.AND P0, PT, R20, 0xdf000000, PT ;            // P0 = (R20 >= 0xdf000000) -- the start of base address
 0x10d0:        ISETP.LT.U32.AND P0, P0, R20, 0xdf200000, P0 ;            // P0 = P0 && (R20 < 0xdf200000 = 0xdf000000 + 2MB)
 0x10e0:        ISETP.EQ.U32.AND P0, P0, R21, 0x7fff, P0 ;                // P0 = P0 && (R21 == 0x7fff) (The higher part has to match)
 0x10f0:  @P0 RET.ABS.NODEC R20 0x0 ;                                     // if (P0) return to (R21 << 32) | R20 (Return only if the conditions above hold)
 0x1100:        BPT.TRAP 0x1 ;                  // Trigger a Trap exception that cause a kernel crash due to invalid return outside the code page
```

Listing 4.2: SASS instrumentation for relative and absolute RETs (Listing 4.1). Each RET is replaced with a branch to a verification code that validates the target address: relative returns check R2 against the section size (0x2980 in this example), and absolute returns validate R20/R21 against the 2MB code page bounds.

```
...
0240:  LDL R20, [R1+0xc] ; //RetAddr loaded from the stack (R1 is the stack pointer)
0250:  BMOV.32 B6, R2 ;
02a0:  IADD3 R1, R1, 0x18, RZ ;
02b0:  RET.ABS.NODEC R20 0x0 ; // unsafe return instruction (RetAddr: R20+R21)
```

Listing 4.3: Unsafe RET.ABS instruction in cudaMalloc for Ampere (SM_80), identified in libcudadevrt.a.

```
...
06a0:  LDL R20, [R1] ; // Load lower part of the RetAddr from the stack
06b0:  BMOV.32 B6, R51 ;
06c0:  BMOV.32 B7, R57 ;
06d0:  LDL R21, [R1+0x4] ; // Load higher part of the RetAddr from the stack
06e0:  IADD3 R1, R1, 0x8, RZ ;
06f0:  RET.ABS.NODEC R20 0x0 ; // unsafe return instruction (RetAddr: R20+R21)
```

Listing 4.4: Unsafe RET.ABS instruction in magmablas_dsort_svd for Ampere (SM_80), identified in libmagma.so [25].

Unsafe RET Instructions. We define a RET instruction as *unsafe* if its target operand is derived from mutable memory or from a value transitively computed from mutable memory, and therefore may be corrupted by an adversary with a memory-write primitive. This definition covers both relative and absolute return modes. For a relative return, such as RET.REL R_i, the instruction is unsafe if the offset register R_i can be loaded from local or global memory, or computed from a value loaded from such memory, because corrupting this offset can redirect control outside the intended function or section. For an absolute return, such as RET.ABS R_i, the instruction is unsafe if any register contributing to the full target address can be loaded from mutable memory, because corrupting that register can redirect the PC to an attacker-controlled data or code page. Conversely, we classify a return as safe when the target is derived only from immutable sources, such as immediate constants, compiler-generated constants, or values loaded from read-only constant memory, and no intervening instruction introduces a dependency on mutable memory. This classification is intentionally conservative: some instructions classified as unsafe may not be exploitable in a particular program context, but every such instruction represents a point where memory corruption can influence the PC unless additional range checks are enforced.

Listing 4.3 shows an example of an unsafe absolute return found in NVIDIA's precompiled libcudadevrt library: the lower part of the return target is restored from local memory before being consumed by RET.ABS. Listing 4.4 presents an additional example of an unsafe absolute return from libMAGMA [25], a widely used open-source GPU linear algebra library for heterogeneous CPU–GPU systems that is extensively employed in scientific computing and high-performance computing (HPC) workloads [7]. These examples illustrate that unsafe returns are not an artifact of our toy exploits or of closed-source CUDA libraries: they also occur in optimized open-source CUDA software, albeit at a lower frequency than in debugging builds.

4.4 SASS-level Instrumentation Instructions

Given the ELF section size, a relative return target can be validated by checking that its offset is non-negative and smaller than the section size, ensuring control remains within the code section. Since code and data pages are mapped to randomized yet mutually relative locations in global memory, larger offsets may cross section boundaries. For absolute returns, validation requires checking that the target address lies within the page bounds (i.e., ≥ the base address and < base + page size). The key challenge is therefore twofold: obtaining the required runtime information (e.g., randomized base addresses and section sizes) and selecting suitable SASS instructions to replace RET with a branch to a verification routine that performs range checks and either resumes execution or raises an exception.

Building on the limited public documentation of the SASS ISA [8, 24, 27], we identify the branch instruction (BRA) to redirect returns to our verification routine. Leveraging nvdisasm [52] and cuobjdump [48], the two common disassembler tools for CUDA binaries, we further identify two additional instructions to enable our cuDEP mechanism: the predicate-setting compare instruction ISETP, which supports 32-bit and 64-bit relational/equality checks

via predicate registers (P0–P7), and `BPT.TRAP`, which raises exception `0x1` (SIGTRAP) and terminates the warp or context. Listing 4.2 illustrates their use in enforcing boundary checks in cuDEP. Appendix C demystifies these instructions in detail for interested readers.

4.5 Obtaining Runtime Information

Similar to NVIDIA's approach in its profiling tools [51], our enforcement of the W⊕X policy relies on interposing selected interactions between the CUDA runtime and driver to extract the required runtime information, as illustrated in Figure 4. To clarify the role of our *Interposer and Instrumentation Plugin*, Figure 5 zooms in on its interaction with the CUDA runtime and highlights the points at which we obtain memory- and code-layout information.

When a CUDA application is compiled, NVCC (or the corresponding compiler) inserts small constructor functions into the resulting ELF binary. At program startup, these constructors invoke `cudaRegisterFatBinary` (followed by `cudaRegisterFunction` for each kernel) to register the FATBIN (PTX/cuBIN bundle) with the CUDA runtime. This registration provides metadata to the runtime without allocating GPU memory or initializing a CUDA context.

Considering the `CUDA Runtime API` (left path in Figure 5), the first device-dependent call, such as `cudaMalloc`, triggers initialization of a primary context on the selected GPU. On Linux, the CUDA runtime communicates with the GPU kernel driver via character devices such as `/dev/NVIDIActl`, `/dev/NVIDIA0`, and `/dev/NVIDIA-uvm`. Our experimental analysis reveals that the runtime interacts with the driver primarily by issuing `ioctl` calls on these device files, which we identify via `strace` and driver inspection. For instance, to service an allocation request, the runtime links against `libcuda.so`, opens these device files, and issues `ioctl` calls to the kernel driver's Resource Manager (RM). These calls allocate GPU memory objects, back them with VRAM pages, and install GPU page-table entries. The returned handle is a 64-bit pointer in the Unified Virtual Address (UVA) space, which resembles a host pointer but is valid for device access unless explicitly mapped into host memory.

Internally, the `CUDA Runtime API` [38] is layered on top of the `CUDA Driver API` [37]. For instance, `cudaMalloc` ultimately invokes `cuMemAlloc`, while context initialization relies on functions such as `cuInit` and `cuDevicePrimaryCtxRetain`. Applications that use the Driver API directly must invoke these functions explicitly. Notably, this layering extends to other operations as well. For instance, `cudaRegisterFatBinary` maps to `cuModuleLoad`, and `cudaLaunchKernel` maps to `cuLaunchKernel`.

Finally, FATBIN registration entails extracting the relevant `cuBIN` image(s) or JIT-compiling selected `PTX` blocks into `cuBIN`s. The runtime parses the resulting ELF objects in host memory and then loads and maps the machine code into the GPU memory via `cuModuleLoad`. Kernel execution follows through `cudaLaunchKernel`, which internally calls `cuLaunchKernel` to dispatch the (instrumented) binary to the relevant GPU compute unit.

4.5.1 Interposing `ioctl` *Calls.* We employ dynamic function interposition via the `LD_PRELOAD` mechanism to inject our plugin (runtime library) into the host-side ELF binary at startup. The plugin overrides selected CUDA runtime entry points (Figure 5) and, within these wrappers, intercepts the underlying `ioctl` invocations

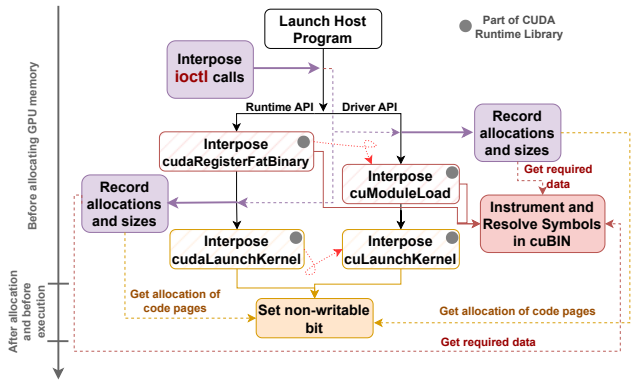

Figure 5: Load-time W⊕X enforcement via CUDA runtime–driver interposition.

to extract the information required for our mitigation. By analyzing the resulting `ioctl`-mediated communication sequence, we observe that, **on Ampere GPUs, only the 15th and 17th allocation requests map executable instructions.** The 15th allocation corresponds to a 2MB page holding the device-side runtime, which includes internal core libraries (analogous to glibc on the CPU), while the 17th allocation maps another 2MB page containing the target kernels and any linked external libraries (e.g., math libraries). Intercepting these allocations enables us to recover the corresponding post-ASLR base addresses (Figure 5) by capturing the relevant `ioctl` parameters before `cuModuleLoad` is invoked. Although the `CUDA Runtime API` ultimately dispatches to `CUDA Driver API` functions (dashed red arrows in Figure 5), interposing only at the `Driver API` boundary is insufficient: when the `Runtime API` calls into the driver stack internally, the relevant `ioctl`s may not be exposed through the `Driver API` symbols visible to our plugin. We therefore support interposition at both API layers.

4.6 Enforcement of cuDEP

To enforce cuDEP, we perform *load-time* SASS instrumentation as cuBINs are registered and loaded by the CUDA software stack. Concretely, our runtime plugin interposes `cudaRegisterFatBinary` or `cuModuleLoad`, depending on whether the application uses the Runtime or Driver API, to locate the FATBIN in host memory. Since the FATBIN format is parsed exclusively by the proprietary runtime, we partially analyze its header to recover its internal structure (see Appendix D for details). Using this knowledge, we enumerate all embedded `cuBIN` modules. For each module, immediately before it is loaded (e.g., via `cuModuleLoad`), we extract code-relevant metadata such as registers usage for return addresses, section sizes, and offsets. When combined with the post-ASLR base addresses of code pages recovered via `ioctl` interposition (Section 4.5), this information is sufficient to instantiate and apply cuDEP checks.

For each FATBIN item, we extract the corresponding `cuBIN` by decompressing it and JIT-compiling PTX code, when needed. We then scan the SASS to identify potentially unsafe indirect control transfers and instrument them accordingly. For instance, we instrument only returns whose targets may be attacker-controlled, such as cases where the return address is spilled to local memory.

Once an unsafe site is identified, we locate the containing code section and inject a branch to our verification routine, together with the necessary range-check logic. In practice, cuBIN code sections include padding to satisfy alignment constraints. On Ampere GPUs, we observe section alignment to 256-byte boundaries. We repurpose these padded NOP regions to host the injected sequence and, when additional space is required, extend the section while restoring alignment by inserting NOPs. Finally, since instrumentation is applied directly to machine code, insertion operates at the instruction-encoding level. Therefore, we implement a custom assembler that emits the required SASS encodings based on our recovered instruction formats (Details are in Appendix C).

4.7 Setting the Non-Writable Bit of Code Pages

Given that code pages are allocated in the 15^{th} and 17^{th} requests to the allocator, and that we can retrieve the randomized base address of each page, enforcing no-write permissions becomes straightforward if applied at the correct point in execution. We therefore set the non-writable bit immediately before the kernel launch. Specifically, just before the allocated and mapped kernel code is executed, we interpose on `cudaLaunchKernel`/`cuLaunchKernel` and issue two `ioctl` calls per page: first unmapping the page via `cuMemUnmap`, and then remapping it without write permission using `cuMemMap`.

5 Portability Across GPU Architectures

Although the preceding sections use Ampere as a representative case study, our findings are not specific to that architecture. To assess the generalizability of both the attacks and the proposed defense, we evaluated the two toy attack variants on Hopper and Blackwell GPUs and found that both succeeded without modification. This observation suggests that the fundamental properties enabling the attacks, including the overall GPU execution model, the absence of hardware-enforced DEP, and the practical behavior of ASLR and stack canaries, remain largely consistent across recent NVIDIA architectures. Our analysis did reveal minor architectural differences in the virtual-address layout, such as changes in the fixed-bit patterns of the base addresses of different memory regions and variations in the relative offsets between them. However, these differences do not qualitatively alter the attack surface and do not affect the applicability of our defense.

The primary differences across Ampere, Hopper, and Blackwell arise at the engineering level rather than the design level. At the binary level, the overall organization of PTX, cuBIN, and SASS remains sufficiently similar for the same load-time instrumentation strategy to apply across architectures, although certain instruction encodings differ. For example, we observed that the encoding of the BRA instruction on Hopper and Blackwell differs from the encoding used on Ampere in our initial prototype. Supporting these architectures therefore required extending our custom assembler to emit the appropriate architecture-specific encodings for the SASS instructions used by the instrumentation logic. At the runtime level, we also observed variations in the sequence of allocation requests responsible for mapping executable GPU regions, depending on both the GPU architecture and the CUDA runtime version. For instance, in our setup, the executable mappings corresponding to the 15^{th} and 17^{th} allocation requests on Ampere shift to the 20^{th} and 22^{nd} on Hopper, and the 30^{th} and 32^{nd} on Blackwell. In practice, accommodating these differences is straightforward and only requires maintaining a small compatibility table within the "interposer and instrumentation plugin" that maps GPU architectures and CUDA runtime versions to the corresponding allocation indices. This table can be incrementally updated as new architectures and runtime versions are introduced.

6 Implementation

Hardware and Software Specifications. We implemented and evaluated our toy attacks and W⊕X prototype on the three NIVIDA architectures discussed in this work:

- A workstation equipped with an `Intel Core i9-10940X CPU`, an `NVIDIA GeForce RTX 3090 GPU` based on the **Ampere** architecture, and 256 GB of system memory. The host system ran `Ubuntu 22.04` on top of `Linux kernel 6.8`. We used `CUDA Runtime v12.8` and `NVIDIA driver v570.133.07`.
- A server with 64 `AMD EPYC 9135 16-Core CPUs`, four `NVIDIA H200 NVL GPUs` based on the **Hopper** architecture, and 770 GB of system memory. The host system ran `Rocky Linux 9.7` on top of `Linux kernel 5.14`. We used `CUDA Runtime v12.8` and `NVIDIA driver v590.48.01`.
- A `NVIDIA DGX Spark` workstation powered by a `NVIDIA GB10 Grace Blackwell Superchip` based on the **Blackwell** architecture, and 128 GB of system memory. The host system ran `NVIDIA DGX OS v7.5.0` on top of `Linux kernel 6.17 (aarch64)`. We used `CUDA Runtime v12.9` and `NVIDIA driver v580.142`

Prototype Details. Our open-source prototype is implemented in C++ and consists of three main components:

- **Assembler.** A custom assembler that emits binary encodings for selected SASS instructions compatible with the target GPU. While cuDEP primarily relies on three instructions, the assembler supports a broader subset of the ISA whose binary encodings we derived during the design phase. Specifically, it supports `RET`, `ISETP`, `TRAP`, `BRA`, `LOP3`, `NOP`, `LD`, `LDL`, `LDG`, `LDC`, `IADD_R`, and `IADD_I`, together with their operand variants. This component comprises approximately 840 lines of code (LOC), excluding headers.
- **cuBIN Parser and Instrumenter.** A module that parses cuBIN ELF objects, handles compressed FATBINs, JIT-compiles PTX to SASS, and injects the cuDEP instrumentation logic into the extracted code structures. This module comprises ≈2060 LOC.
- **Interposer.** A runtime plugin that interposes on CUDA runtime `ioctl` system calls to extract allocation records and base addresses. It also implements the logic for setting the non-writable bit on code pages. This component comprises ≈500 LOC.

7 Evaluation

Goal. The objective of our evaluation is to quantify the performance impact of deploying our defense by measuring runtime, load-time, and memory overheads, while also highlighting its security benefits.

Benchmarks. We selected a diverse set of publicly available benchmarks that are either commonly used in prior GPU security research [28, 29, 76] or openly distributed. Our selection spans a broad range of application domains, programming models, and

workload characteristics. For each benchmark, we executed the default dataset provided with its distribution.

- **Rodinia** [4]: A set of 17 applications developed to capture several computational patterns emulating real-world GPU workloads.
- **Parboil** [3]: A collection of 8 computation-intensive applications developed to compare CPU and GPU performance.
- **GraphBIG** [1]: A benchmark suite for graph computing with 15 applications derived from IBM System G use cases, covering several graph analytics scenarios.
- **Chai** [20]: A set of 14 applications targeting integrated heterogeneous architectures to evaluate CPU–GPU collaboration patterns.
- **Hetero-Mark** [73]: A suite of 3 applications from domains such as signal processing, machine learning, and bioinformatics, designed to evaluate synchronization, memory management, and load balancing in CPU–GPU collaborative computing.
- **XSBench** [78]: A single proxy application that abstracts the main kernel in Monte Carlo reactor simulations, providing a workload with realistic computational intensity.
- **CUDA Samples** [44]: A collection of 25 applications distributed by NVIDIA, covering a broad range of CUDA kernels.
- **llama3.cuda** [60]: A pure CUDA implementation of the LLaMA-3 model, designed for efficient LLM inference on a single GPU.
- **CALM** [2]: A CUDA-accelerated LLM inference framework optimized for maximum single-GPU utilization with a lightweight implementation (evaluated using the Mistral-7B-Instruct model).
- **YALM** [5]: A minimal C++/CUDA implementation of LLM inference with no external dependencies except for weight handling, also evaluated using the Mistral-7B-Instruct model.

Methodology. To quantify memory overhead, we utilize the `cuobjdump` disassembler [48] together with standard Linux utilities (e.g., `size`) to measure the size increase in the `.text` section. We further characterize each workload by computing the fraction of indirect control-transfer instructions and identifying the subset deemed unsafe and thus instrumented.

We define load-time overhead as the interval between host process start and GPU code deployment. This interval is measured using a custom monitoring module that records elapsed time (in milliseconds via kernel timers) from process initialization until the invocation of `cudaRegisterFatBinary`.

For runtime overhead, we rely on the NVIDIA Nsight Compute profiler [51], which supports two complementary measurement modes: elapsed kernel execution time and the number of executed instructions. We report both metrics in Table 5. The elapsed-time metric captures end-to-end kernel latency, but is more sensitive to runtime noise and transient system effects; by contrast, the instruction-count metric provides a more stable view of the overhead introduced by our instrumentation and, in our experiments, yields less noisy measurements. Each experiment is repeated 100 times with and without our defense to reduce variance and improve measurement reliability.

7.1 Performance Evaluation

In the remainder of this section, we report results for the selected benchmarks compiled with `-O3` to reflect real-world deployments and to assess the prevalence of unsafe indirect control-transfer instructions under aggressive optimization.

7.1.1 Code Size Overhead. Tables 2, 3, and 4 report the code-size overhead of `cuDEP` for benchmarks compiled with `-O3` on Ampere, Blackwell, and Hopper, respectively. Overall, the additional overhead remains small across all three architectures. On Ampere, the average increase in the size of the `.text` section is **0.157%**, with a maximum of **0.778%** in `llama3`. On Blackwell, the average increases slightly to **0.175%**, with a maximum of **0.614%**, again in `llama3`. Hopper exhibits the highest overhead among the three architectures, but it remains modest overall, with an average of **0.291%** and a maximum of **1.504%** in `llama3`. Setting the non-writable bit for code pages does not contribute to this overhead, as it only updates page-table permissions at load time.

While base counts naturally vary due to partially differing ISAs, register spilling, and compiler heuristics, the variation across benchmarks and architectures is largely explained by the number of unsafe, and therefore instrumentable, control-flow transfer instructions, together with the availability of alignment `NOP`s that can be reused to host injected checks. Even under `-O3`, potentially unsafe `RET` instructions remain limited but non-negligible. As shown, we identify **125** such instructions on Ampere, **94** on Blackwell, and **93** on Hopper, while unsafe non-`RET` control-flow transfers are rare or entirely absent. Nevertheless, a benchmark with a relatively high fraction of instrumented sites, such as `llama3`, can still incur a more visible increase in code size. Conversely, when sufficient alignment `NOP`s are available, our instrumentation can overwrite them without further expanding the containing section; otherwise, additional `NOP`s must be inserted to preserve section alignment.

The middle sections of these tables further report the instruction-level overhead of our instrumentation, both including and excluding `NOP`s. Excluding `NOP`s, the average increase in non-`NOP` SASS instructions is **0.266%** on Ampere, **0.346%** on Blackwell, and **0.423%** on Hopper. Thus, even when measured at the granularity of non-padding instructions only, the expansion introduced by `cuDEP` remains well below 1% on average across all evaluated architectures. Multi-application benchmarks are reported as single entries in all three tables.

7.1.2 Runtime Overhead. Table 5 reports the runtime overhead of our W⊕X policy across Ampere, Hopper, and Blackwell for the selected benchmarks. Similar to the code-size results, we attribute the runtime overhead exclusively to the `cuDEP` instrumentation, as enforcing non-writable code pages does not incur measurable cost during kernel execution. We report two complementary runtime metrics using NVIDIA Nsight Compute: executed-instruction overhead and elapsed execution-time overhead.

Overall, the runtime overhead remains very low across all three architectures. When measured in terms of executed instructions, the average overhead is only **0.0270%** on Ampere, **0.0152%** on Hopper, and **0.0525%** on Blackwell. The corresponding execution-time overheads remain modest as well, averaging **0.0510%**, **0.1604%**, and **0.3267%**, respectively. At the benchmark level, the largest instruction-count overhead is observed in `Chai` on Ampere (**0.0913%**) and Blackwell (**0.2429%**), and in `Rodinia` on Hopper (**0.0908%**). When measured by elapsed time, the maximum overhead reaches **0.1750%** on Ampere (`llama3`), **0.874%** on Hopper (`llama3`), and **0.8079%** on Blackwell (`Cuda Samples`).

Table 2: Memory overhead introduced by cuDEP on benchmarks compiled with -O3 on Ampere.

Benchmark	Size of the .text section (Bytes)			SASS Instructions (#)			Non-NOP SASS Instructions (#)			RET (#)		Non-RET CFT (#)	
	Original	Instrumented	Overhead(%)	Original	Instrumented	Overhead(%)	Original	Instrumented	Overhead(%)	Total	Unsafe	Safe	Unsafe
Rodinia	1031936	1032704	0.07442%	64496	64544	0.07442%	63664	63774	0.17278%	40	22	4	0
Parboil	78592	78592	0%	4912	4912	0%	4672	4682	0.21404%	4	2	0	0
GraphBIG	97536	97664	0.13123%	6096	6104	0.13123%	5656	5666	0.17680%	2	2	0	0
Cuda Samples	2237824	2238080	0.01144%	139864	139880	0.01144%	133636	133714	0.05837%	87	13	0	7
Chai	175872	175872	0%	10992	10992	0%	10766	10771	0.04644%	9	1	0	0
Hetero-Mark	13568	13568	0%	960	960	0%	907	907	0%	0	0	0	0
XSBench	1192320	1195136	0.23618%	74520	74696	0.23618%	73752	73992	0.32541%	60	48	0	0
llama3	32896	33152	0.77821%	2056	2072	0.77821%	2000	2015	0.75000%	8	3	0	0
calm	447488	449024	0.34325%	27968	28064	0.34325%	27794	27944	0.53968%	32	30	11	0
yalm	89984	89984	0%	5624	5624	0%	5383	5403	0.37154%	12	4	0	0
Total		**Average:**	**0.157%**		**Average:**	**0.157%**		**Average:**	**0.266%**	**254**	**125**	**15**	**7**

Table 3: Memory overhead introduced by cuDEP on benchmarks compiled with -O3 on Blackwell.

Benchmark	Size of the .text section (Bytes)			SASS Instructions (#)			Non-NOP SASS Instructions (#)			RET (#)		Non-RET CFT (#)	
	Original	Instrumented	Overhead(%)	Original	Instrumented	Overhead(%)	Original	Instrumented	Overhead(%)	Total	Unsafe	Safe	Unsafe
Rodinia	915456	917248	0.19575%	57216	57328	0.19575%	56594	56724	0.22971%	36	26	0	0
Parboil	77312	77440	0.16556%	4832	4840	0.16556%	4617	4627	0.21659%	5	2	0	0
GraphBIG	115072	115072	0%	7192	7192	0%	6698	6703	0.07465%	2	1	0	0
Cuda Samples	6650624	6651904	0.01925%	415664	415744	0.01925%	408967	409057	0.02201%	51	15	0	0
Chai	152448	152832	0.25189%	9528	9552	0.25189%	9300	9325	0.26882%	9	5	0	0
Hetero-Mark	17024	17024	0%	1064	1064	0%	1032	1032	0%	0	0	0	0
XSBench	609024	609536	0.08407%	38064	38096	0.08407%	37493	37548	0.14669%	12	11	0	0
llama3	41728	41984	0.61350%	2608	2624	0.61350%	2530	2570	1.58103%	8	8	0	0
calm	514304	515328	0.19910%	32144	32208	0.19910%	31995	32080	0.26567%	17	17	0	0
yalm	113408	113664	0.22573%	7088	7104	0.22573%	6857	6902	0.65626%	12	9	0	0
Total		**Average:**	**0.175%**		**Average:**	**0.175%**		**Average:**	**0.346%**	**152**	**94**	**0**	**0**

Table 4: Memory overhead introduced by cuDEP on benchmarks compiled with -O3 on Hopper.

Benchmark	Size of the .text section (Bytes)			SASS Instructions (#)			Non-NOP SASS Instructions (#)			RET (#)		Non-RET CFT (#)	
	Original	Instrumented	Overhead(%)	Original	Instrumented	Overhead(%)	Original	Instrumented	Overhead(%)	Total	Unsafe	Safe	Unsafe
Rodinia	589440	591104	0.28230%	36840	36944	0.28230%	36222	36352	0.35890%	36	26	0	0
Parboil	75776	75904	0.16892%	4736	4744	0.16892%	4493	4508	0.33385%	4	3	0	0
GraphBIG	101376	101376	0%	6336	6336	0%	5842	5852	0.17117%	2	2	0	0
Cuda Samples	2200832	2201472	0.02908%	137552	137592	0.02908%	131760	131838	0.05920%	49	13	0	0
Chai	158848	159104	0.16116%	9928	9944	0.16116%	9699	9709	0.10310%	9	2	0	0
Hetero-Mark	13696	13696	0%	856	856	0%	826	826	0%	0	0	0	0
XSBench	1387136	1387776	0.04614%	86696	86736	0.04614%	85950	86010	0.06981%	12	12	0	0
llama3	34048	34560	1.50376%	2128	2160	1.50376%	2063	2103	1.93892%	8	8	0	0
calm	470528	471296	0.16322%	29408	29456	0.16322%	29239	29324	0.29071%	17	17	0	0
yalm	92160	92672	0.55556%	5760	5792	0.55556%	5537	5587	0.90302%	12	10	0	0
Total		**Average:**	**0.291%**		**Average:**	**0.291%**		**Average:**	**0.423%**	**149**	**93**	**0**	**0**

Table 5: Runtime and load-time overhead of W⊕X across several benchmarks on latest NVIDIA GPU architectures.

Benchmark	Blackwell O3			Hopper O3			Ampere O3		
	Runtime (inst)	Runtime (time)	Loadtime	Runtime (inst)	Runtime (time)	Loadtime	Runtime (inst)	Runtime (time)	Loadtime
Rodinia	0.0504%	0.4938%	1.6726%	0.0908%	0.2044%	1.2126%	0.08059%	0.0173%	2.1978%
Parboil	0.0000%	0.1257%	1.0645%	0.000001%	0.0250%	0.1751%	0.0000%	0.013%	0.1339%
GraphBIG	0.0932%	0.2291%	1.1480%	0.0170%	0.1237%	1.0996%	0.0659%	0.1434%	0.1416%
Cuda Samples	0.1214%	0.8079%	1.5760%	0.0205%	0.0885%	1.6545%	0.02121%	0.0780%	0.5285%
Chai	0.2429%	0.7249%	0.5922%	0.0079%	0.0694%	0.2330%	0.0913%	0.0358%	0.3663%
Hetero-Mark	0.0000%	0.1110%	2.3295%	0.00%	0.0507%	0.6321%	0.0000%	0.0025%	1.4694%
XSBench	0.0000%	0.0230%	1.3305%	0.00%	0.0155%	1.3105%	0.0000%	0.0040%	1.6736%
llama3	0.0006%	0.6348%	1.7693%	0.00082%	0.874%	0.2509%	0.0006%	0.1750%	0.782%
calm	0.0124%	0.0358%	3.1108%	0.013%	0.012%	0.4662%	0.01094%	0.00510%	0.977%
yalm	0.0044%	0.0807%	3.4552%	0.002%	0.141%	3.5828%	0.000005%	0.0350%	0.761%
Average	**0.0525%**	**0.3267%**	**1.8049%**	**0.0152%**	**0.1604%**	**1.0893%**	**0.0270%**	**0.0510%**	**0.9032%**

The low runtime overhead is explained by the limited use of indirect forward-edge control transfers in CUDA binaries and the large register files available per thread, which reduce the need to spill return addresses to the stack. Consequently, only a small number of control flow instructions per application are instrumented under aggressive optimization, which keeps the dynamic cost of cuDEP small across all evaluated architectures.

7.1.3 Load-time Overhead. Table 5 also reports the load-time overhead of our W⊕X policy across Ampere, Hopper, and Blackwell for -O3 binaries. This is the only overhead category in which both cuDEP instrumentation and the enforcement of correct code-page

permissions contribute, although the effect is dominated by the former. Overall, the load-time overhead remains moderate, with average overheads of **0.9032%** on Ampere, **1.0893%** on Hopper, and **1.8049%** on Blackwell.

At the benchmark level, the maximum load-time overhead is observed in `Rodinia` on Ampere (**2.1978%**), in `yalm` on Hopper (**3.5828%**), and in `yalm` on Blackwell (**3.4552%**). More generally, load-time overhead correlates with the amount of binary rewriting performed by our plugin, including the number of instrumented instructions, the number of embedded `cuBIN`/FATBIN objects that must be processed, and the extent to which sections must be rewritten or extended to preserve alignment. This explains why the load-time overhead varies more noticeably across benchmarks and architectures than runtime overhead, even though it remains low in absolute terms and is incurred only once before execution.

7.2 Security Evaluation

Unfortunately, there is no publicly available CUDA benchmark suite for evaluating memory corruption vulnerabilities and their potential exploitation. Benchmarks used in prior NVIDIA research [76] are proprietary and therefore unsuitable for our study. To evaluate the effectiveness of our W⊕X policy, we reproduced the two attack variants described in Section 3.3 with the policy enabled. Both attacks were successfully blocked, confirming the effectiveness of our approach.

8 Related Work

To the best of our knowledge, our proposal is the first attack-centric defense designed to thwart code injection attacks on modern NVIDIA GPUs. Apart from the Compute Sanitize suite provided by NVIDIA [43], which offers tools to detect race conditions, out-of-bounds memory accesses, uninitialized memory and other issues, existing academic work has primarily focused on vulnerability-centric defenses, particularly addressing out-of-bounds reads and writes. These efforts have typically been realized either as pre-deployment sanitizers [76] or as runtime exploit mitigations [16, 28, 29, 62, 71], mainly through bounds-checking mechanisms [16, 29, 62].

However, the growing body of research exposing weaknesses and potential attack vectors in accelerators [11, 21, 30, 32, 35, 61, 63, 64, 70, 74, 80, 84–86, 88], with NVIDIA GPUs in particular receiving significant attention as the most widely deployed devices for accelerating ML workloads [23, 35, 61, 63, 67, 85, 88], highlights the need to adopt basic defenses, such as a W⊕X policy, which are already mature and enabled by default in all major CPU architectures. Importantly, lessons from over three decades of CPU security research demonstrate that vulnerability-centric runtime mitigations (e.g., bounds checkers [10, 40, 41]) have not seen wide deployment due to prohibitive performance overheads or intrusive designs requiring hardware modifications [22, 75]. A similar pattern emerges in the current, though limited, body of bounds-checking research for NVIDIA GPUs. Existing proposals either incur substantial runtime overheads [28, 76, 83] or require hardware modifications that are unlikely to be adopted [29, 83]. In contrast, as demonstrated in this paper, basic defenses like W⊕X can be enforced with minimal and acceptable overheads. Moreover, the cost could be further reduced if NVIDIA were to reintroduce architectural support for a Never-eXecute (NX) bit to enable data execution prevention, as was available in previous generations. (see Table 1).

Binary Instrumentation Frameworks. To date, NVBIT [79] is the only publicly available framework that supports binary instrumentation of CUDA binaries. However, we could not leverage it for two reasons. First, NVBIT is closed source and exposes a limited API that does not provide the functionality required by our defense. In particular, it offers no way to set the non-writable bit on code pages, implying that instrumented code would remain writable. Second, NVBIT performs instrumentation dynamically at runtime, which incurs substantial performance overhead and is therefore unsuitable for real-world deployments. In contrast, our approach instruments binaries at load time, avoiding per-instruction runtime costs while enabling precise control over memory permissions.

9 Discussion

A Note on Library Sizes. Although NVIDIA GPUs natively support at least three page sizes, 4KB, 64KB, and 2MB, CUDA APIs only allow allocations with a minimum page size of 2MB [81], effectively making 2MB the default page size. We did not encounter any benchmark application whose generated `cuBIN` image fully occupies this space with all linked library functions as well as our instrumentation running on the GPU device. While the host application can be configured to run multiple `cuBIN` images within the same context, as in the `XSBench` application [78] (see Section 7.1.3), we observed that these images are first loaded into host memory by the CUDA runtime and then lazily loaded onto the GPU device as needed. This process involves unmapping the previous code page and remapping it with the new `cuBIN` image. Since our technique for setting code pages as non-writable operates analogously, it fully supports this lazy loading mechanism and does not introduce any compatibility or functional issues.

Other GPUs. We also investigated whether similar mechanisms or gaps have been documented for GPUs from other vendors, including AMD and Intel. However, we were unable to identify publicly available information detailing their support for W⊕X, DEP-like mechanisms, or the handling of executable permissions in GPU memory. This lack of transparency suggests that understanding the security posture of other GPU architectures will require direct access to these platforms and experimental analysis of their software and hardware stacks, which we leave for future work.

10 Conclusion

This work highlights the need to bring fundamental security guarantees, exemplified by the W⊕X policy, to GPUs and other accelerators, where such protections remain largely absent despite being standard on CPUs. We demonstrated that code injection can succeed on modern GPUs even with ASLR and stack canaries enabled, exposing a tangible gap in existing defenses. At the same time, we showed that enforcing an efficient, robust, and lightweight W⊕X policy is feasible in practice, despite challenges such as the absence of a hardware NX bit and the proprietary nature of the GPU software stack. While our prototype currently represents the only practical mechanism for enforcing W⊕X on existing NVIDIA GPUs, we view it not as a final solution but as a call to action: future GPU architectures should reintroduce architectural support for DEP and actively enforce W⊕X at the GMMU level.

Acknowledgments

This research is partially funded by the Internal Funds KU Leuven, by the Cybersecurity Research Program Flanders, and by the EU DUCA Project GA: 101086308.

Open Science

In support of open science, we will provide the full artifact of our defense prototype and toy demonstration attacks, together with instructions and scripts for reproducing our evaluation results, at https://github.com/CarloRamponi/cuWoX, once the conference paper is published.

Ethical Considerations

This research was conducted on a commercial product developed by a major hardware vendor. To minimize potential harm, we deliberately avoided demonstrating attacks on real-world applications and instead relied on controlled, synthetic examples containing intentionally introduced vulnerabilities. This approach ensured that no production systems or user data were exposed during our study. Throughout our work, we also adhered to the vendor's usage and licensing guidelines by avoiding any direct modification to proprietary components of the software stack [49].

In line with responsible disclosure practices, we reported our findings to NVIDIA on August 27, 2025. On November 7, 2025, the relevant NVIDIA security team responded, confirming that they had validated the reported issues and were actively working on corresponding mitigations. They further requested access to our full manuscript and the accompanying artifact to facilitate their internal evaluation. On November 19, 2025, we received a final communication from NVIDIA acknowledging awareness of the identified security gap, which they attributed to "known design limitations". Given that the primary objective of our work is to address this gap, NVIDIA indicated that they would track our contribution as a "security hardening enhancement".

References

[1] -. [n. d.]. A Comprehensive Benchmark Suite for Graph Computing. https://github.com/graphbig/graphBIG [Online; accessed 22-April-2025].

[2] -. [n. d.]. CUDA accelerated language model inference. https://github.com/zeux/calm [Online; accessed 22-April-2025].

[3] -. [n. d.]. GPU Parboil Benchmark. https://github.com/yuhc/gpu-parboil [Online; accessed 22-April-2025].

[4] -. [n. d.]. GPU Rodinia Benchmark. https://github.com/yuhc/gpu-rodinia [Online; accessed 22-April-2025].

[5] -. [n. d.]. Yet Another Language Model. https://github.com/andrewkchan/yalm [Online; accessed 22-April-2025].

[6] 2003. OpenBSD 3.3. OpenBSD Project release notes. https://www.openbsd.org/33.html Introduces W$\hat{X}$ ("W xor X") as a fine-grained memory permissions policy.

[7] Ahmad Abdelfattah, Natalie Beams, Robert Carson, Pieter Ghysels, Tzanio Kolev, Thomas Stitt, Arturo Vargas, Stanimire Tomov, and Jack Dongarra. 2024. MAGMA: Enabling Exascale Performance with Accelerated BLAS and LAPACK for Diverse GPU Architectures. *The International Journal of High Performance Computing Applications* 38, 5 (2024), 468–490. doi:10.1177/10943420241261960

[8] Hamdy Abdelkhalik, Yehia Arafa, Nandakishore Santhi, and Abdel-Hameed A Badawy. 2022. Demystifying the nvidia ampere architecture through microbenchmarking and instruction-level analysis. In *2022 IEEE High Performance Extreme Computing Conference (HPEC)*. Ieee, 1–8.

[9] Advanced Micro Devices, Inc. 2025. *AMD64 Architecture Programmer's Manual, Volume 2: System Programming*. Advanced Micro Devices, Inc. https://docs.amd.com/api/khub/documents/sD1_QL~h4Afq2_tvzxqqSQ/content Publication No. 24593, Rev. 3.43 (documents NX/NXE page protection).

[10] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. 2009. Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors.. In *USENIX Security Symposium*, Vol. 10. 96.

[11] Android Red Team and Arm Product Security Team. 2024. Google & Arm - Raising The Bar on GPU Security. https://security.googleblog.com/2024/09/google-arm-raising-bar-on-gpu-security.html [Online; accessed 22-April-2024].

[12] Authors. 2026. cuWoX: Source Code and Evaluation Artifacts. https://github.com/CarloRamponi/cuWoX.

[13] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[14] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*.

[15] William J Dally, Stephen W Keckler, and David B Kirk. 2021. Evolution of the graphics processing unit (GPU). *IEEE Micro* 41, 6 (2021), 42–51.

[16] Bang Di, Jianhua Sun, Dong Li, Hao Chen, and Zhe Quan. 2018. GMOD: A dynamic GPU memory overflow detector. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*. 1–13.

[17] Emmanuel Ohiri. [n. d.]. A beginner's guide to NVIDIA GPUs. https://www.cudocompute.com/blog/a-beginners-guide-to-nvidia-gpus [Online; accessed 01-June-2025].

[18] Lukas Giner, Roland Czerny, Christoph Gruber, Fabian Rauscher, Andreas Kogler, Daniel De Almeida Braga, and Daniel Gruss. 2024. Generic and Automated Drive-by GPU Cache Attacks from the Browser. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*. 128–140.

[19] giorgiore27. 2023. CUDA Errors Determinte Sticky-ness. https://forums.developer.nvidia.com/t/cuda-errors-determine-sticky-ness/271625 [Online; accessed 13-October-2025].

[20] Juan Gomez-Luna, Izzat El Hajj, Victor Chang, Li-Wen Garcia-Flores, Simon Garcia de Gonzalo, Thomas Jablin, Antonio J Pena, and Wen-mei Hwu. 2017. Chai: Collaborative Heterogeneous Applications for Integrated-architectures. In *Performance Analysis of Systems and Software (ISPASS), 2017 IEEE International Symposium on*. IEEE.

[21] Google Project Zero. 2020. Attacking the Qualcomm Adreno GPU. https://googleprojectzero.blogspot.com/2020/09/attacking-qualcomm-adreno-gpu.html [Online; accessed 22-April-2024].

[22] Google SEcurity Blog. [n. d.]. Securing tomorrow's software: the need for memory safety standards. https://security.googleblog.com/2025/02/securing-tomorrows-software-need-for.html [Online; accessed 22-April-2025].

[23] Yanan Guo, Zhenkai Zhang, and Jun Yang. 2024. GPU Memory Exploitation for Fun and Profit. In *33rd USENIX Security Symposium (USENIX Security 24)*. 4033–4050.

[24] Ari B Hayes, Fei Hua, Jin Huang, Yanhao Chen, and Eddy Z Zhang. 2019. Decoding CUDA binary. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 229–241.

[25] Innovative Computing Laboratory. 2026. MAGMA: Matrix Algebra on GPU and Multi-core Architectures. https://icl.utk.edu/magma/. Accessed: 2026-05-01.

[26] Intel Corporation 2023. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1*. Intel Corporation. https://cdrdv2-public.intel.com/812386/253668-sdm-vol-3a.pdf Order Number: 253668-082US (documents execute-disable / NXE support).

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

[28] Jaewon Lee, Euijun Chung, Saurabh Singh, Seonjin Na, Yonghae Kim, Jaekyu Lee, and Hyesoon Kim. 2025. Let-Me-In:(Still) Employing In-pointer Bounds Metadata for Fine-grained GPU Memory Safety. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 1648–1661.

[29] Jaewon Lee, Yonghae Kim, Jiashen Cao, Euna Kim, Jaekyu Lee, and Hyesoon Kim. 2022. Securing gpu via region-based bounds checking. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 27–41.

[30] Sangho Lee, Youngsok Kim, Jangwoo Kim, and Jong Kim. 2014. Stealing webpages rendered on your browser by exploiting GPU vulnerabilities. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 19–33.

[31] Lei Mao. 2022. Proper CUDA Error Checking. https://leimao.github.io/blog/Proper-CUDA-Error-Checking/ [Online; accessed 11-October-2025].

[32] Chris S Lin, Joyce Qu, and Gururaj Saileshwar. 2025. GPUHammer: Rowhammer Attacks on GPU Memories are Practical. *arXiv preprint arXiv:2507.08166* (2025).

[33] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437* (2024).

[34] Microsoft. 2022. Data Execution Prevention. https://learn.microsoft.com/en-us/windows/win32/memory/data-execution-prevention [Online; accessed 13-February-2023].

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

[36] Modal.com. [n. d.]. Host Software. https://modal.com/gpu-glossary/host-software [Online; accessed 01-May-2025].

[37] Modal.com. 2025. CUDA Driver API. https://modal.com/gpu-glossary/host-software/cuda-driver-api [Online; accessed 01-May-2025].

[38] Modal.com. 2025. CUDA Runtime API. https://modal.com/gpu-glossary/host-software/cuda-runtime-api [Online; accessed 01-May-2025].

[39] Aaftab Munshi, Benedict Gaster, Timothy G Mattson, and Dan Ginsburg. 2011. *OpenCL programming guide*. Pearson Education.

[40] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. 2009. SoftBound: Highly compatible and complete spatial memory safety for C. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 245–258.

[41] Myoung Jin Nam, Periklis Akritidis, and David J Greaves. 2019. FRAMER: a tagged-pointer capability system with memory safety applications. In *Proceedings of the 35th Annual Computer Security Applications Conference*. 612–626.

[42] Nvidia. [n. d.]. A RoadMap to Nvidia Architectures. https://www.nvidia.com/en-us/technologies/ [Online; accessed 01-August-2025].

[43] Nvidia. [n. d.]. Compute Sanitizer. https://docs.nvidia.com/compute-sanitizer/ComputeSanitizer/index.html [Online; accessed 22-April-2025].

[44] Nvidia. [n. d.]. CUDA Samples. https://github.com/NVIDIA/cuda-samples [Online; accessed 22-April-2025].

[45] Nvidia. [n. d.]. NVIDIA Linux Open GPU Kernel Module Source. https://github.com/NVIDIA/open-gpu-kernel-modules/tree/main/kernel-open/nvidia-uvm [Online; accessed 22-April-2025].

[46] Nvidia. 2025. Cuda Binary Utilities. https://docs.nvidia.com/cuda/archive/9.0/pdf/CUDA_Binary_Utilities.pdf [Online; accessed 22-April-2025].

[47] NVIDIA. 2025. CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/ [Online; accessed 22-April-2025].

[48] NVIDIA. 2025. cuobjdump. https://docs.nvidia.com/cuda/cuda-binary-utilities/index.html [Online; accessed 22-April-2025].

[49] Nvidia. 2025. End User License Agreement. https://docs.nvidia.com/cuda/eula/index.html [Online; accessed 01-August-2025].

[50] NVIDIA. 2025. Memory Management. https://docs.nvidia.com/cuda/cuda-for-tegra-appnote [Online; accessed 01-May-2025].

[51] Nvidia. 2025. Nsight Compute CLI. https://docs.nvidia.com/nsight-compute/NsightComputeCli/index.html [Online; accessed 22-April-2025].

[52] NVIDIA. 2025. nvdisasm. https://docs.nvidia.com/cuda/cuda-binary-utilities/index.html [Online; accessed 22-April-2025].

[53] Nvidia. 2025. nvFatbin. https://docs.nvidia.com/cuda/nvfatbin/index.html [Online; accessed 22-April-2025].

[54] Nvidia. 2025. Nvidia CUDA Toolkit Documentation. https://docs.nvidia.com/cuda/archive/12.4.0/cuda-runtime-api/group__CUDART__DEVICE.html [Online; accessed 01-May-2025].

[55] Nvidia. 2025. Parallel Thread Execution ISA Version 9.0. https://docs.nvidia.com/cuda/parallel-thread-execution/ [Online; accessed 01-August-2025].

[56] Nvidia. 2025. Pascal MMU Format Changes. https://nvidia.github.io/open-gpu-doc/pascal/gp100-mmu-format.pdf [Online; accessed 22-April-2025].

[57] Nvidia. 2025. Tuning CUDA Applications for NVIDIA Ampere GPU Architecture. https://docs.nvidia.com/cuda/ampere-tuning-guide/index.html [Online; accessed 22-April-2025].

[58] NVIDIA Corporation. 2026. CUDA Driver API: Module Management. https://docs.nvidia.com/cuda/cuda-driver-api/group__CUDA__MODULE.html. Accessed: 2026-05-07.

[59] NVIDIA Corporation. 2026. CUDA Virtual Memory Management. https://docs.nvidia.com/cuda/cuda-programming-guide/04-special-topics/virtual-memory-management.html. Accessed: 2026-05-07.

[60] Sang Park. 2024. llama3.cuda: pure C/CUDA implementation for Llama 3 model. https://github.com/likejazz/llama3.cuda llama3.cuda, MIT License.

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

[62] Manos Pavlidakis, Giorgos Vasiliadis, Stelios Mavridis, Anargyros Argyros, Antony Chazapis, and Angelos Bilas. 2024. Guardian: Safe GPU sharing in multi-tenant environments. In *Proceedings of the 25th International Middleware Conference*. 313–326.

[63] Roberto Di Pietro, Flavio Lombardi, and Antonio Villani. 2016. CUDA leaks: a detailed hack for CUDA and a (partial) fix. *ACM Transactions on Embedded Computing Systems (TECS)* 15, 1 (2016), 1–25.

[64] Frederik Dermot Pustelnik, Xhani Marvin Sass, and Jean-Pierre Seifert. 2024. Whispering Pixels: Exploiting Uninitialized Register Accesses in Modern GPUs. In *2024 IEEE 9th European Symposium on Security and Privacy (EuroS&P)*. IEEE, 345–360.

[65] Donald Ray and Jay Ligatti. 2012. Defining code-injection attacks. *Acm Sigplan Notices* 47, 1 (2012), 179–190.

[66] Jonas Roels, Adriaan Jacobs, Silviu Vlasceanu, Mahmoud Ammar, and Stijn Volckaert. 2026. From Prompt To Pwn: Exploiting GPU Memory Errors During ML Inference. In *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[67] Jonas Roels, Adriaan Jacobs, and Stijn Volckaert. 2025. CUDA, Woulda, Shoulda: Returning Exploits in a SASS-y World. In *Proceedings of the 18th European Workshop on Systems Security*. 40–48.

[68] Sihyun Roh, Woohyuk Choi, Jaeyoung Chung, Yoochan Lee, Suhwan Song, and Byoungyoung Lee. 2025. GHost in the Shell: A GPU-to-Host Memory Attack and Its Mitigation. (2025).

[69] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. 2011. Q: Exploit hardening made easy. In *20th USENIX Security Symposium (USENIX Security 11)*.

[70] Tyler Sorensen and Heidy Khlaaf. 2024. Leftoverlocals: Listening to LLM Responses through Leaked GPU Local Memory. *arXiv preprint arXiv:2401.16603* (2024).

[71] Michael B Sullivan, Mohamed Tarek Ibn Ziad, Aamer Jaleel, and Stephen W Keckler. 2023. Implicit memory tagging: No-overhead memory safety using alias-free tagged ecc. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*. 1–13.

[72] Yifan Sun. 2011. What are the parameters for cudaRegisterFatBinary and cudaRegisterFunction? Stack Overflow. https://stackoverflow.com/questions/6392407/what-are-the-parameters-for-cudaregisterfatbinary-and-cudaregisterfunction-f Accessed: 2025-08-27.

[73] Yifan Sun, Xiang Gong, Kavyan Amir, Amir Kavyan Ziabari, Leiming Yu, Xiangyu Li, Saoni Mukherjee, Carter McCardwell, Alejandro Villegas, and David Kaeli. 2016. Hetero-Mark, A Benchmark Suite for CPU-GPU Collaborative Computing. doi:10.1109/IISWC.2016.7581262

[74] Ariel Szabo and Uzy Hadad. 2025. Crypto Miner Attack: GPU Remote Code Execution Attacks. *arXiv preprint arXiv:2502.10439* (2025).

[75] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*. IEEE, 48–62.

[76] Mohamed Tarek Ibn Ziad, Sana Damani, Aamer Jaleel, Stephen W Keckler, and Mark Stephenson. 2023. Cucatch: A debugging tool for efficiently catching memory safety violations in cuda applications. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 124–147.

[77] Philippe Tillet, H. T. Kung, and David Cox. 2019. Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL '19)*. doi:10.1145/3315508.3329973

[78] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. 2014. XS-Bench - The Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis. In *PHYSOR 2014 - The Role of Reactor Physics toward a Sustainable Future*. Kyoto. https://www.mcs.anl.gov/papers/P5064-0114.pdf

[79] Oreste Villa, Mark Stephenson, David Nellans, and Stephen W Keckler. 2019. Nvbit: A dynamic binary instrumentation framework for nvidia gpus. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 372–383.

[80] Yingchen Wang, Riccardo Paccagnella, Zhao Gang, Willy R Vasquez, David Kohlbrenner, Hovav Shacham, and Christopher W Fletcher. 2024. GPU.zip: On the Side-Channel Implications of hardware-based Graphical Data Compression. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 3716–3734.

[81] Jinhui Wei, Jianzhuang Lu, Qi Yu, Chen Li, and Yunping Zhao. 2020. Dynamic gmmu bypass for address translation in multi-gpu systems. In *IFIP International Conference on Network and Parallel Computing*. Springer, 147–158.

[82] Yujie Liu. 2023. CUDA Runtime Error and Restore. https://blog.yujieliu.com/blogs/hpc/kernel-restore.html [Online; accessed 11-October-2025].

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

[84] Yicheng Zhang, Ravan Nazaraliyev, Sankha Baran Dutta, Andres Marquez, Kevin Barker, and Nael Abu-Ghazaleh. 2025. NVBleed: Covert and Side-Channel Attacks on NVIDIA Multi-GPU Interconnect. *arXiv preprint arXiv:2503.17847* (2025).

[85] Zhenkai Zhang, Tyler Allen, Fan Yao, Xing Gao, and Rong Ge. 2023. 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*. 960–974.

[86] Zhe Zhou, Wenrui Diao, Xiangyu Liu, Zhou Li, Kehuan Zhang, and Rui Liu. 2016. Vulnerable gpu memory management: towards recovering raw data from gpu. *arXiv preprint arXiv:1605.06610* (2016).

[87] Ruofan Zhu, Ganhao Chen, Wenbo Shen, Lyuye Zhang, Dakun Shen, Rui Chang, and Yanan Guo. 2025. Demystifying and Exploiting ASLR on NVIDIA GPUs. In *Proceedings of the 47th IEEE Symbosium on Security and Privacy (S&P)*.

[88] Zhiting Zhu, Sangman Kim, Yuri Rozhanski, Yige Hu, Emmett Witchel, and Mark Silberstein. 2017. Understanding the security of discrete GPUs. In *Proceedings of the General Purpose GPUs*. 1–11.

A Address Layout and Offsets under ASLR on Ampere GPUs

Table 6: Observed offsets between selected GPU memory spaces for a representative CUDA application on Ampere GPUs, compiled with *O0* and executed with ASLR enabled. The table summarizes offsets measured across 40,000 executions. All offsets are 32 bits, except for the offset between global and user-defined constant memory, which is 16 bits. Some memory-space pairs exhibit a single fixed offset value (e.g., the offset between global and constant memory is always 0x100), whereas others show a small set of possible offset values. In the latter case, variations occur exclusively in the most significant byte of the 32-bit offset, implying at most 2^8 distinct offset values. These results were verified with an additional 1,000 executions using the *O3* optimization flag, yielding similar behavior with only slight differences in the starting addresses of some memory spaces.

Memory Spaces	Observed Offsets
Global - Local	0x0FBFFA68 0x11BFFA68 0x13BFFA68
Global - Shared	0x10BFFF00 0x12BFFF00 0x14BFFF00
Global - Constant	0x100
Local - Constant	0x0FBFFB68 0x11BFFB68 0x13BFFB68
Shared - Constant	0x10C00000 0x12C00000 0x14C00000
Local - Shared	0x01000498

Table 7: Starting addresses of selected GPU memory spaces for a representative CUDA application on Ampere GPUs, compiled with *O0* and executed with ASLR enabled. The table reports 10 representative runs out of 40,000 total executions. All memory spaces exhibit randomized base addresses with fixed relative offsets between them, except for heap allocations issued from within device code, which remain unrandomized.

Run	Global	Local	Shared	Constant	Heap (cudaMalloc)	Device Heap
1	0x78e4fc400100	0x78e50bfffb68	0x78e50d000000	0x78e4fc400000	0x78e4fc200000	**0x2055ff920**
2	0x769a7c400100	0x769a8bfffb68	0x769a8d000000	0x769a7c400000	0x769a7c200000	**0x2055ff920**
3	0x7664e4400100	0x7664f7fffb68	0x7664f9000000	0x7664e4400000	0x7664e4200000	**0x2055ff920**
4	0x7f0920400100	0x7f092ffffb68	0x7f0931000000	0x7f0920400000	0x7f0920200000	**0x2055ff920**
5	0x74b13c400100	0x74b14ffffb68	0x74b151000000	0x74b13c400000	0x74b13c200000	**0x2055ff920**
6	0x7287f4400100	0x728807fffb68	0x728809000000	0x7287f4400000	0x7287f4200000	**0x2055ff920**
7	0x78cb88400100	0x78cb9bfffb68	0x78cb9d000000	0x78cb88400000	0x78cb88200000	**0x2055ff920**
8	0x7593a0400100	0x7593affffb68	0x7593b1000000	0x7593a0400000	0x7593a0200000	**0x2055ff920**
9	0x72b460400100	0x72b46fffffb68	0x72b471000000	0x72b460400000	0x72b460200000	**0x2055ff920**
10	0x7363ac400100	0x7363bfffffb68	0x7363c1000000	0x7363ac400000	0x7363ac200000	**0x2055ff920**

B An Overview of Unsafe Branch Instructions in CUDA Libraries

In Table 8, we analyze all libraries shipped with the CUDA toolkit to assess the prevalence of potentially unsafe indirect branch instructions. Such instructions update the program counter (PC) using register operands having their values loaded from memory and thus might be controlled by adversaries in the case of memory corruption. By leveraging the limited available documentation of SASS for older GPU architectures, along with experimental analysis for the Ampere GPU architecture [24, 27], we identify three relevant indirect branch instructions: `BRX` (Branch relative indirect), `JMX` (Branch absolute indirect), and `CALL R*` (Indirect function call). For every device function, we classify occurrences of these instructions as safe if the target register is derived from either (i) a `LDC` instruction (load from constant memory, assumed safe as it is read-only), or (ii) a `MOV RN, 0x*` (move immediate; hardcoded and therefore immutable considering our W⊕X policy in place). Every other case is considered unsafe. This classification is conservative, meaning some instructions classified as unsafe might actually be safe (non-exploitable).

Important Note: Libraries reporting 0 functions or instructions do not contain any `device code`. Instead, they serve as host-side interfaces that issue calls to device functions, whose implementations are supplied by other device-side libraries.

Table 8: Analysis of Safe (S) and Unsafe (U) Indirect Branch Instructions in Major CUDA Libraries from NVIDIA

Library	# Functions	# Instructions	BRX		JMX		CALL R*		RET	
			S	U	S	U	S	U	S	U
libcublasLt	12431	6144728	4	0	0	0	0	0	158	2489
libcublas	4564	2776704	0	0	0	0	0	0	390	1240
libcudadevrt.a	126	29392	0	0	0	0	0	0	7	87
libcudart	0	0	0	0	0	0	0	0	0	0
libcufft	9	1904	0	0	0	0	0	16	1	0
libcufftw	0	0	0	0	0	0	0	0	0	0
libcufile_rdma	0	0	0	0	0	0	0	0	0	0
libcufile	4	336	0	0	0	0	0	0	0	0
libcufilt.a	0	0	0	0	0	0	0	0	0	0
libculibos.a	0	0	0	0	0	0	0	0	0	0
libcupti	0	0	0	0	0	0	0	0	0	0
libcurand	296	251368	0	0	0	0	0	0	27	277
libcusolver_lapack	71	66088	0	0	0	0	0	0	4	42
libcusolver_metis	0	0	0	0	0	0	0	0	0	0
libcusolver	2743	2437888	0	0	0	0	202	0	394	2495
libcusparse	4710	5264848	1	0	0	0	0	0	2304	3135
libmetis	0	0	0	0	0	0	0	0	0	0
libnppc	0	0	0	0	0	0	0	0	0	0
libnppial	1789	281272	55	0	0	0	0	0	155	509
libnppicc	521	87272	10	0	0	0	0	0	11	42
libnppidei	1158	150048	23	0	0	0	0	0	0	124
libnppif	2766	1685080	13	0	0	0	0	0	44	251
libnppig	1446	676648	0	0	0	0	0	0	53	839
libnppim	229	154808	12	0	0	0	0	0	0	0
libnppist	1641	732528	4	0	0	0	0	0	84	771
libnppisu	0	0	0	0	0	0	0	0	0	0
libnppitc	516	83344	316	0	0	0	0	0	0	0
libnpps	1088	253184	176	0	0	0	0	0	36	227
libnvfatbin	0	0	0	0	0	0	0	0	0	0
libnvJitLink	0	0	0	0	0	0	0	0	0	0
libnvjpeg	243	65208	0	0	0	0	0	0	21	18
libnvperf_host	0	0	0	0	0	0	0	0	0	0
libnvptxcompiler	0	0	0	0	0	0	0	0	0	0
libnvrtc-builtins	0	0	0	0	0	0	0	0	0	0
libnvrtc	0	0	0	0	0	0	0	0	0	0
libcuda	0	0	0	0	0	0	0	0	0	0
Total	**36351**	**21142648**	**614**	**0**	**0**	**0**	**202**	**16**	**3689**	**12546**

C SASS Instructions Breakdown

To develop our protection mechanisms, we needed to operate directly at the binary instruction level. This required not only analyzing existing SASS instructions but also inserting newly assembled ones into the program stream. Since NVIDIA's SASS instruction set

architecture (ISA) is proprietary and closed-source, only minimal descriptions of the mnemonics are publicly available. No official documentation of the binary encoding exists. Therefore, we were forced to reverse engineer the encoding of selected instructions by systematically observing changes in the assembly representation under `nvdisasm` while flipping bits in the encoded instruction.

The following diagrams summarize our findings. They illustrate the internal structure of several instructions, showing how opcodes, predicates, registers, immediates, and control fields are mapped to bits within the instruction bytes.

An instruction is 16 bytes (128 bits) long and contains both the encoded instruction and the control codes used by the hardware to schedule instructions more efficiently [27]. As shown in Figure 6, control codes are 24 bits long and they occupy the most significant bits (MSB) of the instruction, leaving 104 bits for the actual instruction encoding.

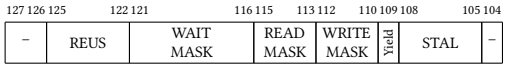

Figure 6: Control codes (24 bits)

Let us now only consider the 104 bits encoding the effective instruction. Every instruction has the opcode and predicate in the 16 least significant bits (LSB), as shown in Figure 7.

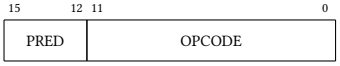

Figure 7: Instruction's opcode and predicate value

Opcodes. The 12-bit opcode is a fixed value that encodes an instruction or instruction class. In Table 9, we list some of the opcodes encoding instructions we reversed as they were needed for analysis or instrumentation.

Mnemonic	Opcode
RET	100101010000
BPT.TRAP	100101011100
BRA	100101000111
ISETP	100000001100

Table 9: Known Ampere SASS opcodes

Predicates. Predicates are special registers that can be used to selectively execute instructions (and also as source and destination register for logic operations). There are 7 registers, from $P0$ to $P6$ and PT (should be $P7$, but is always set at 1). They are encoded in the instruction using 4 bits, where $0 \leq n \leq 6$ corresponds to P_n, 7 encodes PT, $8 \leq n \leq 14$ corresponds to $!P_{n-8}$ and finally 15 encodes $!PT = PF$. In Listing 4.2, on lines 10 and 17 you can see two predicate instructions, that will only be executed by the threads whose $P0 = 1$. All the other instructions hold predicate PT and thus it is omitted from the SASS assembly.

Instruction Body. Figures 8 to 11 show instruction encodings, from what we have been able to infer by selectively flipping the instruction bits and looking at the disassembled SASS instruction

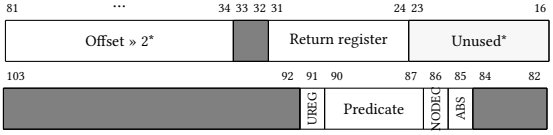

Figure 8: RET instruction on $80 \leq SM < 90$ (Ampere). For $SM \geq 90$ (from Hopper), the first 8 *Unused** bits encode the LSB of offset >> 2 and bits 34-81 encode offset >> 10

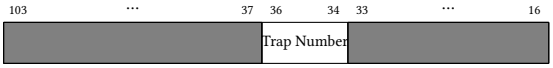

Figure 9: BPT.TRAP instruction

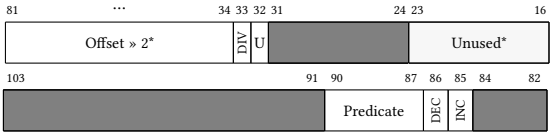

Figure 10: BRA instruction on $80 \leq SM < 90$ (Ampere). For $SM \geq 90$ (from Hopper), the first 8 *Unused** bits encode the LSB of offset >> 2 and bits 34-81 encode offset >> 10

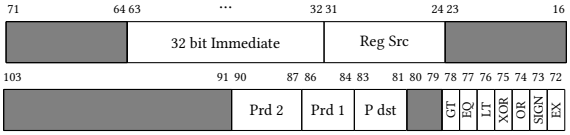

Figure 11: ISETP instruction

from `nvdisasm`. The blocks in **gray** mark bits that are unused or have an unknown behavior.

The implementation of an assembler for these instructions and some other can be found in the tool's source code we have provided, in files `cuInstruction.h/cpp`.

D FATBIN Structure

CUDA executables embed device code ("fat binaries") that the runtime registers at startup via `__cudaRegisterFatBinary`. Since NVIDIA does not document these structures, we reconstructed them based on prior community reverse engineering [72], available sources [53], and our own analysis. Listing D.1 contains the structure of `__fatBinC_Wrapper_t`, which is the handle of a fat binary image, and the only argument of `__cudaRegisterFatBinary` from which our analysis starts. This structure is a wrapper for the real fat binary contents, pointed by the `data` field.

```
1 #define FATBINC_MAGIC   0x466243B1
2 typedef struct {
3  int magic;
4  int version;
5  const unsigned long long* data;
6  void *filename_or_fatbins;
7 } __fatBinC_Wrapper_t;
```

Listing D.1: Fatbin handle definition in `fatbinary_section.h` from the CUDA Toolkit

Listing D.2 shows the internal structure that we have inferred. The pointer in the wrapper structure points to an array of `fat_elf_header`,

identified by magic number `0xBA55ED50`. Each header contains a list of `fat_elf_item` in which every item contains a cuBIN file (compiled SASS) or PTX code to be JIT compiled. We have not reverse-engineered every field of the innermost structure, as they were not needed for our protection. The unknown fields are named `param_*`, `type` identifies whether the contained file is a cuBIN or PTX code, `header_size` is the size of the header itself, useful because the source name (last field) has a variable length, and marks where the data starts (`header + header->header_size`). The remaining fields are self-explanatory and can be further studied in our implementation of cuWX.

```
struct fat_elf_header {
  uint32_t magic;
  uint16_t version;
  uint16_t header_size;
  uint64_t size;    };
struct fat_elf_item {
  uint16_t type;
  uint8_t param_2;
  uint8_t param_3;
  uint32_t header_size;
  uint64_t size;
  uint32_t compressed_size;
  uint32_t param_4;
  uint16_t minor;
  uint16_t major;
  uint32_t compute_version;
  uint32_t obj_name_offset;
  uint32_t obj_name_len;
  uint64_t flags;
  uint64_t zero;
  uint64_t uncompressed_size;
  char src_name[];   };
```

Listing D.2: Reverse engineered fat binary structures