From Prompt to Pwn: Exploiting GPU Memory Errors During ML Inference

Jonas Roels
DistriNet, KU Leuven
Ghent, Belgium

Adriaan Jacobs
DistriNet, KU Leuven
Ghent, Belgium

Silviu Vlasceanu
Independent researcher
Munich, Germany

Mahmoud Ammar
Independent researcher
Munich, Germany

Stijn Volckaert
DistriNet, KU Leuven
Ghent, Belgium

Abstract

GPUs accelerate a growing fraction of modern computing workloads. While prior work has studied the exploitation of memory errors in GPU applications written in C/C++ dialects, existing attacks rely on artificially introduced memory errors that grant attackers capabilities that may not arise when exploiting real-world GPU software. Consequently, GPU vendors and developers continue to largely treat such errors as reliability concerns rather than security issues, leaving vulnerabilities unpatched and potentially exploitable.

This paper challenges that assumption. We show, for the first time, that remote, unprivileged adversaries *can* trigger device-side memory errors in realistic ML applications under specific conditions. We develop controlled proof-of-concept exploits that escalate vulnerabilities in the PyTorch ML framework and the CuDF dataframe library into *write-what-where* primitives, and weaponize such errors to enable denial-of-service, targeted manipulation and degradation of ML models, sensitive data leakage, and device-side code injection.

To facilitate exploit construction, we design and implement a dynamic taint-tracking framework for NVIDIA GPUs that tracks attacker-controlled data flows to identify exploitable memory objects. Our findings demonstrate that adversaries can conduct memory-corruption attacks on vulnerable ML inference servers, highlighting the need to reassess prevailing assumptions about GPU memory safety and adopt essential mitigations, such as Write-XOR-eXecute, that are currently absent from modern GPUs.

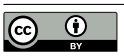
CCS Concepts

• **Security and privacy → Systems security**; **Software reverse engineering**; **Domain-specific security and privacy architectures**.

Keywords

Memory errors, ML inference, GPUs, taint tracking, code injection

ACM Reference Format:
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 '26), November 15–19, 2026, The Hague, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3830454.3846730



CCS '26, The Hague, Netherlands

ACM ISBN 979-8-4007-2871-6/2026/11
https://doi.org/10.1145/3830454.3846730

1 Introduction

Modern software increasingly relies on hardware accelerators, particularly graphics processing units (GPUs), to achieve high performance. New general-purpose GPU (GPGPU) workloads, including database management [3] and machine learning (ML) [30, 74, 82], expose GPUs to interactive use, for example, multi-modal prompts in large-language model inference, often by remote and potentially unauthenticated users. Developers primarily implement GPGPU computations using memory-unsafe programming frameworks, such as NVIDIA CUDA [25] and AMD ROCm [8, 56], which extend C or C++. On CPUs, memory errors in these languages remain a dominant source of critical software vulnerabilities [14, 70, 85, 88]. The widespread adoption of these frameworks has similarly resulted in memory errors across real-world GPU codebases [66, 71, 87], and even in applications written in memory-safe languages such as Python, which rely on GPU libraries implemented in memory-unsafe frameworks [60, 61] or employ unsafe Python-based GPU domain-specific languages such as Triton [100].

Recent work has shown that memory errors in GPU workloads could theoretically be abused to implement many traditional CPU-style exploits, including code-injection [34, 67] and code-reuse attacks [52, 73, 86]. However, the evaluated attacks typically require a high degree of adversarial control over the victim application, and all existing proof-of-concept (PoC) exploits introduce custom memory errors into self-written vulnerable kernels (i.e., device-side functions executed in parallel on the GPU) to demonstrate feasibility. While insightful, it remains unclear whether remote adversaries can gain sufficient control over vulnerabilities in real-world GPU programs to enable such expressive exploitation in real-world environments. After all, adversaries typically interact with GPUs *indirectly* via a host-side inference stack that parses requests and marshals input, thereby restricting exploitability. In addition, many GPU workloads, such as ML inference, are heavily data-oriented and may transform or constrain payload values before they reach vulnerable layers.

Overall, the feasibility of remote exploits has not yet convinced GPU vendors: despite their prevalence (Section 3), we are aware of two CVEs assigned to on-device memory errors [68, 69], with developers assuming "Denial of Service" (DoS) and data leakage as the primary risks. As such, vendors continue to treat GPU memory

errors predominantly as reliability issues rather than security vulnerabilities, and leave them unpatched for months after disclosure.

This paper challenges the status quo. We show, for the first time, that malicious users *can* interact with real-world vulnerable GPU device code with sufficient control to trigger memory errors remotely, provided that the deployment satisfies the vulnerability's triggering conditions. We then demonstrate that adversaries can bootstrap constrained out-of-bounds (OoB) writes into near-arbitrary write-what-where primitives, enabling multi-stage exploitation and leading to DoS, ML model degradation, targeted manipulation, sensitive data leakage, and code injection attacks in a controlled in-process environment. We posed and answered three key research questions:

[Q1] What adversarial capabilities do memory errors in real-world GPU codebases enable?

[Q2] Which GPU memory objects and structures are susceptible to corruption?

[Q3] How can adversaries manipulate the victim program state via memory errors to hijack these targets?

To answer **Q1**, we analyze 116 memory errors across 19 GPU codebases and characterize their cause, how remote adversaries can trigger them, and which device memory regions they affect. For **Q2** and **Q3**, we design and implement a dynamic taint-analysis framework for NVIDIA GPUs atop the NVBIT dynamic binary instrumentation (DBI) engine [101]. Our framework is the first open-source dynamic taint tracker compatible with modern NVIDIA architectures. We tuned it to facilitate GPU-specific memory-error exploit development, e.g., by identifying *bootstrapping gadgets* such as store operations that dereference tainted addresses originating from corruptible device memory.

We empirically validate the feasibility of device-side exploitation by implementing PoC exploits against real device-side vulnerabilities in PyTorch [74] and CuDF [2], popular frameworks for ML inference and dataframe acceleration, respectively. We evaluate these PoCs in a controlled environment that isolates GPU-side exploitability from deployment-specific effects, including network jitter and timeouts, and therefore do not demonstrate end-to-end exploits.

In summary, this paper makes the following contributions:

- We survey publicly disclosed device-side memory errors across popular GPU codebases and characterize them by root cause and potential for remote exploitation.
- We design and implement a new dynamic taint-analysis framework for NVIDIA GPUs and use it to conduct experiments on vulnerability expressiveness, identify bootstrapping gadgets, and facilitate GPU memory error exploitation.
- We demonstrate that adversaries can trigger real device-side memory errors in real ML applications (i.e., PyTorch and CuDF) using API requests under specific deployment characteristics.
- We bootstrap constrained OoB writes into expressive exploitation primitives, showing that such bugs can affect product security, not only reliability.

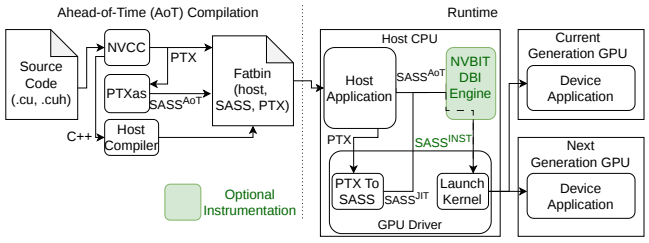


Figure 1: NVIDIA CUDA compilation and execution flow.

We open-source our taint analysis and all PoC exploits at: https://github.com/RoelsJonas/From-Prompt-to-Pwn and https://doi.org/10.5281/zenodo.22671972.

2 Background

This paper focuses on NVIDIA GPUs, the dominant platform for GPU computing. While all technical details reflect NVIDIA's Blackwell architecture [23], we discuss the applicability of our techniques to other architectures in Section 10.2.

NVIDIA GPUs implement the single instruction, multiple threads (SIMT) execution model, in which all threads within a warp, typically 32, execute the same instructions in lockstep. Each streaming multiprocessor (SM) supports the execution of one or more warps and maintains a large register file. During scheduling, the driver allocates up to 256 general-purpose registers and 8 predicate registers per thread, as well as up to 64 general-purpose and 8 predicate uniform registers per warp.

CUDA software developers write GPU kernels in a C++ dialect [25]. CUDA C++ provides built-in variables for thread and block indexing that identify a thread's position within the three-dimensional grid and block hierarchy defined by the developer. Kernels can access various device memory spaces. `Global` memory for data exchange with the host and `local` memory for thread-local state (e.g., the stack) reside in VRAM. `Shared` memory acts as a fast, on-chip scratchpad. Finally, developers, the compiler, and the driver use `constant` memory to store read-only values during kernel execution, including kernel arguments, grid dimensions, and function addresses. While memory-space-specific load and store instructions exist (`LD` or `ST` postfixed with `G/L/S/C`), generic load and store instructions (`LD` and `ST`) can also access global, shared, and local memory by encoding the memory space within the address.

Similar to host processes, CUDA contexts isolate concurrently executing CUDA applications [25]. A host process may manage multiple contexts, each containing one or more streams. Kernels within a single stream execute sequentially, while kernels issued to different streams may execute concurrently on the same GPU.

Developers compile CUDA code using the NVIDIA CUDA Compiler (NVCC). It separates host and device code, passes the host code to a standard compiler (e.g., GCC), and translates the device code into an intermediate representation, called Parallel Thread Execution (PTX). The PTX assembler (PTXas) compiles PTX ahead of time (AoT) into architecture-specific, undocumented Streaming ASSembler (SASS) instructions. To maintain compatibility across GPU generations, NVCC bundles host, PTX, and SASS code into a

FATBIN file. If the installed GPU is incompatible with the bundled SASS code, the driver just-in-time (JIT) compiles the PTX code.

Figure 1 shows this compilation and execution pipeline, including optional DBI via NVBIT [101]. NVBIT instruments binaries at the SASS level before the driver transfers code to device memory, ensuring compatibility with both AoT and JIT compilation. It replaces instructions with calls to instrumentation logic, leveraging the fixed-length instruction encoding of modern NVIDIA GPUs, i.e., 16 bytes for Blackwell [15].

2.1 GPU-Backed Server Deployments

At a high level, GPU-backed servers typically implement a layered software architecture. In ML inference, model developers distribute architectures and pretrained parameters via repositories such as Hugging Face [45], often as Python modules and weight files. Inference frameworks, such as the general-purpose Transformers [44] library and specialized vLLM [62] inference engine, consume these artifacts and handle model loading, request orchestration, and execution policies tailored to specific model classes. These frameworks operate atop tensor runtimes such as PyTorch [74], which translate high-level inference logic into sequences of device operations. Similarly, in data analytics, frameworks such as BlazingSQL [3] and NVIDIA's Spark GPU extension [20] use GPU-accelerated dataframe libraries, such as CuDF [2], to accelerate data ingestion and transformation.

Across these domains, control code invokes host-side wrapper functions that marshal inputs, manage device memory, compute request-dependent metadata such as shapes and strides, and configure GPU kernel launches to execute the core numerical operations. In production, these frameworks are often embedded in serving systems such as NVIDIA Triton [26], which manage request routing, model lifecycles, health checks, and automatic recovery. Thus, if an illegal memory access or other error terminates the CUDA context or the host process, the service restarts and reloads its original state.

The specific kernels executed for a given request depend on the input (e.g., input shape or requested database operation) and on deployment parameters (e.g., the deployed model architecture). Request-dependent parameters propagate through the underlying framework into kernel launch configurations and device-side indexing logic. Thus, two requests to the same endpoint may exercise different kernels (e.g., CuDF's heuristic to pick a concatenation kernel; Section 3.1.2) or invoke the same kernel with different launch parameters, depending on the input.

Although requests are independent at the application level, GPU-backed servers commonly retain long-lived GPU-resident state within a CUDA context. This avoids repeated startup costs, such as model instantiation and host-to-device transfers of large weight tensors, and thereby improves latency and throughput.

3 A Look at Memory Errors in GPU Code

To understand the nature and prevalence of memory errors in GPU software, we conduct a systematic analysis of real-world CUDA vulnerabilities and projects. We identify **103** on-device memory errors across 11 popular GPU projects with at least 1,000 GitHub stars and over 30,000 lines of device code as of December 2025, including PyTorch [74], NVIDIA CCCL [18] and its predecessor CUB [24], RAPIDS AI libraries (CuDF [2], CuML [4], CuVS [5], and RAFT [6]), OpenMM [40], ArrayFire [10], llama.cpp [47], PaddlePaddle [32], and TensorFlow [33] by manually analyzing and deduplicating public issues and pull requests identified using the following keywords: "memory error", "out of bounds", "compute sanitizer", "memcheck", "illegal memory access", "cudaErrorIllegalAddress", and "error 700". We also include **10** device-side memory errors documented in prior academic work [17, 42, 76, 105], covering benchmarks such as Parboil [94], StreamMR [41], Rodinia [13], and Hetero-Mark [96], as well as NVIDIA's NVJPEG image-processing library [27]. Our manual analysis uncovered **3** additional previously unreported memory errors. Two of these errors occur in Parboil and one in PyTorch.

Table 1: Prevalence of spatial GPU memory error categories in our dataset; some errors combine memory operations.

Category	Load	Store	Atomic	Total
Integer overflow	52 (53.6%)	49 (74.2%)	0 (0.0%)	**61** (54.0%)
Incorrect/missing thread-index check	24 (24.7%)	10 (15.2%)	2 (66.7%)	**28** (24.8%)
Incorrect/missing stride check	5 (5.2%)	3 (4.5%)	1 (33.3%)	**6** (5.3%)
Incorrect/missing input validation	4 (4.1%)	1 (1.5%)	0 (0.0%)	**4** (3.5%)
Incorrectly placed check	3 (3.1%)	0 (0.0%)	0 (0.0%)	**3** (2.7%)
Other errors	9 (9.3%)	3 (4.5%)	0 (0.0%)	**11** (9.7%)
Total	**97**	**66**	**3**	**113**

Out of the **116** surveyed memory errors, we observed **113** spatial memory errors, **2** uninitialized reads, and **1** temporal memory error. We categorized the spatial memory errors by their root programming mistake. Table 1 summarizes the results; we provide full details in our artifact.

Integer overflows account for the majority of memory errors in our dataset (54%). These errors arise when an arithmetic operation brings an index or offset outside the bounds of its underlying data type typically a 32-bit signed integer, before being used in pointer arithmetic. While 32-bit offsets historically sufficed for earlier generations of GPUs and their relatively limited memory (i.e., less than 4 GB), modern GPUs support substantially larger memories and datasets. However, many GPU codebases do not reflect this change [93], resulting in a high prevalence of integer overflow vulnerabilities. Listings 1 and 2 illustrate representative real-world instances of this class of errors in PyTorch and CuDF, respectively.

Incorrect or missing thread-index and stride checks form the second and third most prevalent GPU memory error causes. These errors stem from incorrect assumptions about the multidimensional thread grid and data structures used in GPU computation, which complicate correct buffer indexing. Common cases include off-by-one errors, such as omitting the equality case in boundary checks (Listing 3) and incorrect assumptions, e.g., when developers assume that input sizes will always be multiples of the thread block or warp dimensions, and omit the check (Listing 4). Thread-index checks determine whether a given thread should execute a particular portion of a kernel, whereas stride checks govern loop bounds when threads process data in a strided manner. As with integer overflows, these errors are often triggered by malicious input dimensions.

Incorrect or missing input validation constitutes a smaller class of errors in which specific input *values*, rather than input

Legend: buggy code potential fix

```
1 int64_t o_plane, osizeH, osizeW, oh, ow;
2 scalar_t *output, *indices;
3 int64_t offset = o_plane*osizeH*osizeW + oh*osizeW + ow;
4 *(ptr_output + offset) = max;
5 *(ptr_ind + offset) = argmax;
```

Listing 1: PyTorch adaptive max pooling integer overflow [79]

```
1 int64_t i = threadIdx.x + blockIdx.x * blockDim.x;
2 size_t* offset_it; T *output_data, *input_data;
3 while (i < output_size) {
4     output_data[i] = input_data[i - *offset_it];
5     i += blockDim.x * gridDim.x; }
```

Listing 2: Integer overflow in CuDF fused concatenate [28]

```
1 if (tid >= WARP_SIZE)
2     return;
3 sumf = buf_iw[tid];
```

Listing 3: Off-by-one thread-index check in llama.cpp [46]

```
1 if (indexInWarp+TILE_SIZE*tilesToStore < BUFFER_SIZE)
2     buffer[QindexInWarp] =
      ↪ buffer[indexInWarp+TILE_SIZE*tilesToStore];
```

Listing 4: Missing index check in OpenMM [72]

dimensions, cause memory errors, e.g., due to incorrect handling of negative inputs or unchecked input offsets.

Incorrectly placed checks represent a minor category in which bounds checks occur after the corresponding memory access. In our dataset, all such instances result in OoB reads. While these errors do not immediately enable data leakage, they can still cause reliability issues. For example, accesses to unmapped memory can crash the CUDA context, enabling DoS.

Other errors include invalid allocation sizes, failure to handle allocation errors, type confusion, null-pointer dereferences, and incorrect argument passing. Collectively, these represent only 9.7% of the surveyed bugs.

Memory regions. Overall, spatial memory errors overwhelmingly occur in global memory (110 out of 113), and the remaining 3 errors target shared memory.

Observation 1

We observe no memory errors affecting local memory in real-world GPU bug reports, despite prior academic work exploiting deliberately introduced cases [52, 73, 86]. This aligns with observations by NVIDIA researchers [99] and reflects the rare use of local memory.

Takeaway 1

Most on-device memory errors reported in open-source projects are *spatial* memory errors, primarily caused by integer overflows and incorrect or missing checks on thread indices and stride variables.

3.1 Triggering Memory Errors Remotely

We now empirically demonstrate that unprivileged remote adversaries *can* trigger two representative real-world device-side memory errors via valid, carefully crafted inputs. We deliberately select errors stemming from the most prevalent class in our dataset: integer overflow. To the best of our knowledge, no prior work has demonstrated real device-side memory errors being triggered through remote GPU-backed APIs.

3.1.1 ML Inference: PyTorch Adaptive Pooling. First, we target the integer overflow bug in PyTorch's adaptive pooling layer [60, 61], originally reported in January 2025, but remained unresolved as of September 2026. We use production-grade inference servers, including Ray Serve [84] and NVIDIA Triton [26], to create a PoC setup. These systems expose REST API endpoints for ML inference. We implement only minimal Python glue code to parse requests, format responses, and configure server parameters such as batching; the deployed models and all underlying CUDA code remain unmodified and represent a realistic, benign public ML inference deployment.

The vulnerable two-dimensional adaptive pooling layers compute output pointers using 32-bit signed arithmetic. By supplying a batch size that, when multiplied by the pooling output dimensions and hidden size (both outside the adversary's control), exceeds 2^{31} (i.e., the upper bound of a 32-bit signed integer), an adversary can trigger an integer overflow during pointer computation (line 3 in Listing 1). The integer overflow results in n overwritten elements, starting 2^{31} elements below the intended global memory output buffer, where $n = batch\_size \times hidden\_size \times output\_size^2 - 2^{31}$. The resulting memory error constitutes a **non-adjacent, linear, direct out-of-bounds write**: a contiguous OoB write of adversarially influenced size starting from a fixed, non-zero offset relative to the vulnerable buffer triggered by the adversary's input.

Any model that uses adaptive pooling with an output size greater than 1 is potentially vulnerable. However, triggering this error requires that the underlying serving system supports large batch sizes and has sufficient VRAM available. Our analysis of ML models using this layer reveals 9 potentially vulnerable architectures in the widely used libraries Transformers [44] and TorchVision [78] (see Table 2). These models were collectively downloaded more than 470,000 times in December 2025 from the Hugging Face model zoo.

Triggering this overflow requires large batch sizes. In practice, the available GPU memory and model footprint limit the maximum configurable batch size. To quantify feasibility, we compute the minimum batch size required to trigger the exploit by analyzing each model's source code, configuring them with FP16 weights, and attempting exploitation on an NVIDIA RTX 5090 (32 GB) and an NVIDIA H100 (80 GB). As Table 2 shows, multiple models have vulnerable batch sizes that fit the VRAM limits of large GPUs commonly deployed for inference. VRAM forms an upper bound for batch size; the serving system may be configured with a lower limit, preventing exploitation when it falls below the threshold.

3.1.2 GPU Data Processing via CuDF. Our second scenario targets GPU-accelerated data ingestion pipelines. CuDF is a dataframe library commonly used by higher-level GPU-accelerated data processing frameworks [3, 20]. We evaluate a bug in CuDF's fused

Table 2: Summary of model architectures with adaptive pooling output sizes > 1, their required batch size to become vulnerable, and monthly downloads as of December 18, 2025.

Model Architecture	Output Size	Hidden Size	Minimum Batch Size	Monthly Downloads	VRAM for exploit
LayoutLMv2	7	256	171,197	423,364	▲
UperNet‡	6	768	77,673	30,245	●
Beit‡	6	768	77,673	6,122	▲
PerceptionLM‡	16	4,096	2,049	6,060	▲
PvtV2‡	7	768	57,066	5,632	◆
TextNet	2	512	1,048,577	757	●
Data2VecVision‡	6	768	77,673	14	▲
VGG	7	512	85,599	N.A.	◆
AlexNet	6	256	233,017	N.A.	▲

Legend: ‡ Only specific instances ● ≤ 32 GiB ◆ ≤ 80 GiB ▲ > 80 GiB N.A. Non-HF model

concatenate kernel [28, 71], where an integer overflow occurs during striding (line 5 of Listing 2), when the total number of rows across all input tables falls between 2^{30} and 2^{31}. Interestingly, the wrapper code includes integer overflow checks, but these are incomplete and catch only simple cases (i.e., $assert(\sum \#rows < 2^{31})$).

Our experimental setup mimics how BlazingSQL [3] constructs tables backed by CuDF dataframes from (untrusted) CSV input files. We confirm that supplying specially crafted CSV files of sufficient collective length can successfully cause memory corruption of $2 \times \sum \#rows - 2^{31}$ buffer elements starting 2^{31} elements below the intended buffer. Note that CuDF only executes the vulnerable fused concatenation kernel if the operation concatenates five or more tables. As in the PyTorch scenario, this vulnerability yields a **non-adjacent, linear, direct out-of-bounds write** limited to a memory region determined by the overflowed stride computation.

While CuDF developers discovered and patched this error in 2022, CuDF suffered from at least 48 other memory errors stemming from this exact programming mistake, some of which remained in the production code for over two years after the first disclosure.

3.2 Expressiveness of Initial Memory Errors

The memory errors characterized above are sufficient to trigger device-side faults, but they do not immediately provide adversaries with expressive exploitation primitives. In particular, integer overflows, the most common class of on-device memory errors in our dataset, cause pointer arithmetic to wrap around and access memory *far* below the intended output buffer. In practice, blindly triggering such bugs commonly accesses unmapped device memory, terminates the CUDA context, and causes **DoS**. This behavior follows from CUDA's global memory allocation strategy, which places allocations in the available slot at the highest address possible.

Therefore, exploitation beyond DoS requires addressing additional challenges: identifying sensitive values that reside in the corruptible memory region (**Q2** in Section 1), such as model weights, (function) pointers, offsets, or device code, and arranging the device memory layout so that the constrained OoB access overlaps with those targets (**Q3** in Section 1).

Takeaway 2

"Blind" exploitation of GPU memory errors likely crashes before further exploitation is possible. To address this, adversaries will need to influence the device's memory layout, such that OoB accesses overlap with target objects.

4 Deployment and Threat Model

We model a benign but vulnerable GPU-backed server that continuously serves requests, such as ML inference or database queries. In particular, at least one valid endpoint invokes a CUDA kernel that contains a memory corruption vulnerability in the tensor runtime, dataframe library, or other underlying GPU library. We assume that this server and its configuration meet the requirements to trigger this vulnerability, e.g., it allows batched inference with a large maximum batch size and has sufficient VRAM to facilitate this. We also assume that the server has an availability mechanism in place, e.g., using common health checks and automatic recovery mechanisms. The server executes all work in a single, long-lived CUDA context, handles multiple requests concurrently via multiple CUDA streams, and maintains GPU-resident state across requests. These properties can realistically arise from common performance optimizations in GPU-backed deployments (e.g., to avoid loading the same model multiple times).

We consider an unprivileged remote adversary that interacts with this service solely through its exposed API. The adversary can issue concurrent requests and choose input values and shapes within service-imposed limits but *cannot* influence deployment parameters such as the model architecture, execute code directly on the victim, load custom operators, nor allocate GPU memory on it (e.g., by directly calling `cudaMalloc`). We assume that adversarially crafted but semantically valid input supplied via a valid request can trigger at least one device-side OoB write (similar to Section 3.1). We do not rely on any vulnerabilities in the host-side code; all attacks target device-side execution and abuse memory errors within CUDA kernels. We assume the victim deploys the GPU-side mitigations available in NVIDIA's toolchain, i.e., ASLR and stack canaries.

The adversary may replicate the deployed software stack offline for analysis and exploit development. The adversary's goals may include DoS, degradation or manipulation of ML model outputs, leakage of sensitive data, device-side code execution, or broader system compromise. Achieving these goals requires corrupting GPU-resident data that is not directly controlled by input values.

Finally, we consider three levels of adversarial knowledge regarding ML model weights, which can be (partially) subject to intellectual property restrictions:

- **[S1] Full knowledge:** the adversary knows all model weights, as in open-weight deployments.
- **[S2] Partial knowledge:** the adversary knows the weights up to a vulnerable layer, modeling deployments that fine-tune only later layers, or use open-weight preprocessing layers.
- **[S3] No knowledge:** the adversary has no knowledge of the model weights, modeling closed-weight deployments.

5 High-Level Attack Overview

In this paper, we aim to demonstrate that constrained real-world device-side memory errors can, under the right conditions, result in

expressive exploits, such as control-flow hijacking and ML model manipulation, whose feasibility has previously only been demonstrated in theory using powerful self-inserted primitives in artificial programs [34, 52, 67, 73, 86]. Real-world vulnerabilities and interactive GPU deployments exhibit non-trivial constraints that can inhibit such expressive exploitation. For instance, we find that the lack of memory errors affecting local memory (see Section 3), the lack of spilled return addresses, and the scarcity of code pointers stored in global memory (as we show in Section 6.2) make it infeasible to implement these exploits directly using first-stage vulnerabilities such as those described in Section 3.1. Moreover, even when sensitive device-resident data exists, such as ML model parameters or code, its inopportune address, e.g., (far) above vulnerable buffers, makes it impossible to overlap *direct* OoB writes with this data. These constraints make prior one-stage exploitation strategies impractical for real-world GPU memory errors.

Consequently, we follow a multi-stage exploitation strategy. First, we elevate a constrained memory error into a near-arbitrary write-what-where primitive. In particular, we attempt to leverage an initial **direct OoB write** to corrupt a *bootstrapping gadget*, whose subsequent use triggers an ***indirect*** **OoB write**.

As bootstrapping gadgets, we specifically focus on **hijackable stores**: store operations whose operands, i.e., the dereferenced address and written data, depend on values loaded from adversary-corruptible device memory. In normal execution, these stores are benign, as the loaded values are well-formed and, in most cases, not adversary-controlled. Once these values become corrupted, e.g., by a prior OoB write, their dereference effectively becomes adversary-controlled, transforming constrained spatial corruption into near-arbitrary write-what-where behavior. Identifying these gadgets and overlapping *direct* OoB writes with their associated sensitive data is non-trivial.

The remainder of this paper details how to make this process practical. In Section 6, we design and implement a dynamic taint analysis for GPU code that systematically identifies bootstrapping gadgets and supports offline reverse engineering and exploit development. Section 7 explains how to combine these gadgets with direct memory errors to create expressive write primitives. Section 8 demonstrates empirically that this exploitation strategy is feasible and effective under the GPU-side execution conditions of real GPU-accelerated deployments.

6 Dynamic GPU Taint Tracking

Manual analysis of data propagation in GPU applications is challenging due to massive parallelism, heterogeneous memory spaces, and complex compiler and driver interactions. It is even more difficult for closed-source libraries and highly optimized kernels. Hence, manual reverse engineering does not scale to identify bootstrapping gadgets across large GPU codebases systematically. Instead, we design and implement a dynamic taint-tracking framework for CUDA to aid exploit development by tracking adversary-controlled data through device code and identifying bootstrapping gadgets.

6.1 Design Rationale and Implementation

Efficiently storing large amounts of metadata in GPU memory remains challenging. While CPU taint trackers often rely on large, reserved, pageable shadow memory regions, CUDA's non-unified memory management API does not support this paradigm because developers cannot allocate or reserve pageable virtual memory. Additionally, distinguishing device memory regions for generic memory accesses and preventing data races during massively parallel taint manipulations pose additional challenges. To address these challenges, our design combines a purpose-built global memory allocator that implicitly tags addresses with the allocation size to enable efficient in-band metadata storage and retrieval [38, 49] (Section 6.1.2) with explicit taint propagation built via NVBIT's [101] SASS-level DBI of AoT- and JIT-compiled user binaries and closed-source libraries, e.g., cuDNN and cuBLAS (Section 6.1.3).

6.1.1 Taint sources, propagation, and sinks. Users can configure (all or certain) allocation sites, specific address ranges, and OoB memory accesses as taint sources. Taint propagates through arithmetic operations and memory accesses across registers and the various device memory spaces (i.e., global, shared, and local). We adopt explicit taint propagation semantics [31]: an operation's result is tainted only if it depends on tainted input operands. We deliberately avoid implicit control-flow tainting, as it leads to over-tainting and imprecision, which is counterproductive for exploit development.

During gadget identification, the primary taint sinks of interest are the operands of memory store operations and indirect branches. We flag such tainted operations as candidate bootstrapping gadgets. Optionally, we log all tainted operations to support reverse engineering and exploit development.

6.1.2 Metadata management and memory layout. For global memory, we implement a custom allocator inspired by low-fat pointers [38], which encodes allocation sizes directly in the pointer value and augments each allocation with in-band metadata to store taint information. At startup, the allocator reserves multiple virtual address regions corresponding to power-of-two size classes, ranging from 512 B (`cudaMalloc`'s minimum allocation size) to 16 GiB.

During allocation, the allocator overprovisions memory to accommodate a single taint bit for every 4 bytes of data. This granularity is configurable based on analysis needs and available GPU memory. The allocator then computes the smallest power of two exceeding the combined object and metadata size and places the allocation in the corresponding region on a power-of-two boundary, yielding the layout shown in Figure 2, with the object at the start of the slot, metadata at the end, and optional padding in between. Taints are initialized to 0 by default and set to 1 for taint sources. This design avoids large shadow memory regions at the cost of modest per-allocation memory overhead, while enabling fast metadata address computation via simple arithmetic (i.e., three bit shifts, two bitwise AND operations, and three integer additions/subtractions), as illustrated in Figure 3.

For registers, shared memory, and local memory, we rely on shadow memory. While their metadata sizes depend on the number of active threads and blocks, setting an upper bound on these values enables us to allocate fixed-size metadata regions at startup, thereby preventing our taint analysis from interfering with CUDA graph operations. For applications that require nearly all available GPU memory, we allocate these metadata structures in unified memory and rely on on-demand paging to avoid out-of-memory errors.

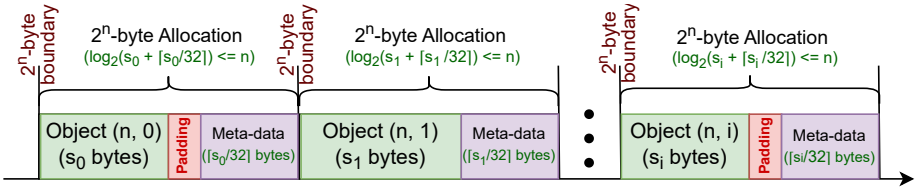


Figure 2: Layout of global memory objects. We add metadata to each object and align the combined size up to the next power of two (n) by padding if necessary.

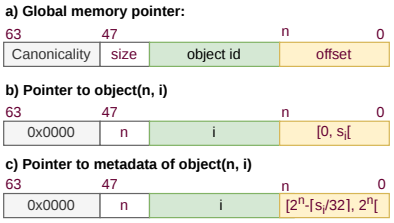


Figure 3: Global mem. pointer layout.

Table 3: Summary of candidate hijackable store gadgets identified (i.e., kernels that each contain one or more hijackable store operations) in PyTorch, CuDF, and their dependencies. Covered Kernels reports the fraction of executed (and thus instrumented) kernels.

Library	Global (STG)		Shared (STS)		Local (STL)		Generic (ST)		Covered
	Instr	Kernels	Instr	Kernels	Instr	Kernels	Instr	Kernels	Kernels
CuDF	700	79	364	29	3	2	541	60	64.1%
PyTorch	387	73	437	19	9	3	3	3	10.8%
CUTLASS	71	2	64	1	0	0	0	0	2.48%
CuBLAS	15	13	0	0	0	0	0	0	2.17%
CUB	1032	85	2166	62	43	5	683	21	17.0%
CuCo	73	10	36	5	0	0	2	1	43.5%
Total	2278	262	3067	116	55	10	1229	85	N.A.

6.1.3 Instrumentation strategy. Upon a kernel's first launch, our custom NVBIT pass analyzes and instruments its SASS code as well as all reachable device functions. Our instrumentation adds taint-propagation logic to arithmetic and memory operations and tracks taint flow across metadata spaces corresponding to each operand. To reduce instrumentation overhead and avoid data races, we carefully organize taint metadata so that adjacent threads access adjacent metadata bytes. This additionally allows warps to fetch taint information using coalesced memory accesses.

6.2 Effectiveness of Gadget Identification

Integer overflows on offset calculations, the root cause of the majority of current CUDA memory errors (see Table 1), generally necessitate careful memory layout manipulation on CUDA to place target objects inside the range of corruptible memory (Section 3.2), instead of simply crashing the program. Therefore, in our second-stage gadget analysis, we consider the entire manipulatable device memory region as a potential corruption target, and leave it to the layout-manipulation techniques in Section 7.1 to place the desired target objects at the corruptible location. This means tainting all global memory objects, and tracking which sensitive operations (indirect branches, memory stores, etc.) have tainted operands (written data *and* dereferenced address for store operations). This is technically an over-approximation, since some parts of global memory (e.g., code pages and other data loaded at startup) are allocated before the GPU kernel first becomes interactive. While we could, in theory, delay the initial taint until the first request arrives to avoid this imprecision, it would require instrumenting thousands of test cases. Therefore, we choose to manually filter the analysis's output.

Table 3 summarizes the candidate hijackable store gadgets identified by our analysis that dereference pointers and write data hijackable by adversaries via other memory errors across all memory types in the test cases of PyTorch and CuDF (i.e., the libraries containing the vulnerabilities triggered in Section 3.1 and exploited in Section 7–8), and their dependencies (CUTLASS, CuBLAS, CUB, and CuCollections). Despite the limited kernel coverage for PyTorch (its test suite covers only a subset of templated kernel instances), we identify many candidate hijackable store gadgets. Importantly, the gadget identification mode does not assess exploitability conditions such as the degree of control or the presence of bounds checks. Instead, it conservatively identifies candidate gadgets.

We validate the functional correctness of our experiments by manually analyzing all **152** candidate kernels (containing at least one "hijackable store") and confirm that all of them indeed dereference tainted pointers. Only **6** of those implement bounds checks on the corruptible indices, preventing the creation of write-what-where primitives. Listings 5 and 6 show representative examples of hijackable store gadgets.

Notably, even after further static analysis covering the remaining kernels, we observe no hijackable indirect branches in PyTorch, CuDF, or their dependencies. Instead, they propagate operands of indirect branches exclusively through registers (e.g., return addresses) and load them from constant memory or use immediate operands (e.g., function addresses). Similarly, we confirm that they do not spill return addresses into (local) memory. Nevertheless, our taint tracker does identify hijackable control-flow transfers in the artificial `vtable` exploit PoCs by Di et al. [34] and Miele [67], demonstrating its ability to surface such gadgets when present.

While the performance overhead and remaining limitations of our taint tracker do not affect its suitability for offline analysis or exploit development, we refer interested readers to our artifact.

Observation 2

We find no indirect branches in CuDF, PyTorch, or their device-side dependencies that adversaries could hijack via memory corruption. Consequently, code-reuse attacks appear substantially constrained in these codebases, leaving code injection as a comparatively more accessible attack vector due to the lack of W⊕X.

7 From Triggers to Primitives: Bootstrapping Memory Errors

Transforming the constrained OoB writes from Section 3.1 into more expressive exploitation primitives, specifically near-arbitrary write-what-where capabilities, requires two ingredients: control

Legend: offset load tainted dereference

```
1 T* topKSliceStart = &topK.data[slice];
2 int64_t* indicesSliceStart = &indices.data[slice];
3 uint32_t startWithinK = withinKCounts[block_idx - 1];
4 uint32_t offset = withinKIndex + startWithinK;
5 topKSliceStart[offset * topKWithinSliceStride] = val;
6 indicesSliceStart[offset * idxWithinSliceStride] = idx;
```

Listing 5: Hijackable stores in PyTorch GatherTopK [80]

```
1 size_type const* d_offsets;
2 char* d_buffer = d_chars + d_offsets[idx];
3 auto const d_str = d_column(idx);
4 for (auto chr : d_str) {
5     if (chr == '\"') d_buffer += string('\"');
6     d_buffer += string(chr); }
```

Listing 6: Hijackable store in CuDF escape_strings [29]

over the device memory layout so that OoB writes target adversary-relevant objects, and benign, but hijackable, store bootstrapping gadgets to amplify these initial corruptions, ultimately yielding write primitives sufficient for advanced exploitation goals.

7.1 Memory Layout Manipulation on GPUs

Bootstrapping exploitation primitives relies on an adversary's ability to control the relative placement of vulnerable and target objects in device memory. Although CUDA does not expose explicit control over allocation addresses, its allocation behavior is sufficiently predictable in practice to enable effective memory layout manipulation [22, 105]. Concretely, reversing CUDA's default global-memory allocator shows that it fills the highest-addressed hole large enough to satisfy the new allocation within its managed region. This allows for manipulation similar to heap grooming, heap feng shui, and heap massaging on CPUs [43, 54, 92].

Figure 4b illustrates one concrete strategy for achieving such layouts. At a high level, the adversary first induces the victim application to allocate a sequence of *manipulation* buffers using benign requests to the victim's ML inference API (e.g., by providing batches of images for classification), advancing the allocator state. They then induce the allocation of the target buffer (i.e., the TopK indices buffer of the GatherTopK hijackable store gadget in Figure 4) via a benign API request. Subsequently, the victim application deallocates earlier manipulation buffers when their corresponding requests complete, creating unallocated space above the target buffer. When a subsequent malicious request induces the vulnerable operation, the allocator places the vulnerable buffer (i.e., the adaptive pooling output buffer in Figure 4) into this hole. This request's malicious input then induces an OoB write during the vulnerable operation, corrupting data at a predictable relative offset. Importantly, this process does not rely on adversaries directly allocating memory nor on memory leaks. Rather, adversaries can utilize valid, concurrent API request sequences that induce framework-managed allocations whose sizes depend on input values (e.g., batch size).

Some GPU libraries, including PyTorch and CuDF, implement their own caching allocators on top of CUDA's allocator to improve performance by avoiding frequent synchronizing operations. While these (often stream-local) caches could hamper memory layout manipulation, inducing large allocations (e.g., $\geq$ 10 MiB), varying buffer sizes, and many subsequent allocations can help avoid them.

7.2 Bootstrapping via Hijackable Stores

Memory layout manipulation alone is insufficient to relocate arbitrary victim data to adversary-chosen addresses. Some sensitive memory contents reside at inaccessible static locations. As illustrated in Figure 4, the CUDA driver places executable code above the region used for dynamically allocated global memory buffers. In addition, buffers allocated during application initialization, such as model weights, may reside at locations unreachable via direct OoB writes. Affecting such regions, therefore, requires adversaries to *bootstrap* direct memory corruption into a more powerful exploitation primitive.

At a high level, bootstrapping proceeds as follows. First, the induced direct OoB write from the previous section corrupts stored pointers or offsets with adversary-controlled values. Later, an otherwise benign kernel scheduled as part of the same benign request that allocated the previous section's target buffer dereferences these corrupted pointers or offsets during a **hijackable store operation**, such that this dereference accesses the otherwise out-of-reach target. This creates a secondary, *indirect* OoB write: a store whose destination address is determined by adversary-controlled data. Note that, generally, hijackable store gadgets are benign under normal conditions. Adversaries can hijack the destination address only via a prior OoB write. Additionally, adversaries cannot directly call kernels containing such gadgets; instead, they induce them via valid ML inference API requests (e.g., to perform inference on a deployed ML model that relies on a hijackable operation). Figure 4c illustrates this amplification step, where the hijackable store operation dereferences the indices stored in the corrupted buffer, thereby transforming the constrained direct OoB write into a write primitive with substantially wider reach.

7.3 Bootstrapping in PyTorch and CuDF

Because manually analyzing highly optimized GPU code to identify hijackable stores is challenging, we rely on our taint tracker (Section 6) to expose candidate store operations influenced by values loaded from writable device memory. Table 3 summarizes these operations across PyTorch, CuDF, and their dependencies.

We empirically validate the feasibility of memory-error bootstrapping using two representative hijackable-store kernels identified by our taint analysis: PyTorch's gatherTopK (Listing 5) and CuDF's escape_strings (Listing 6).

7.3.1 PyTorch: gatherTopK as a bootstrapping gadget. The gatherTopK kernel loads per-slice offsets from a device-resident buffer (withinKCounts) and uses them to index stores into output arrays (Listing 5, lines 3–6). These offsets are 32-bit unsigned integers. Once adversaries corrupt the offset buffer, they can redirect stores across a wide range of indices relative to the intended output base. Depending on the element type, this provides control over writes

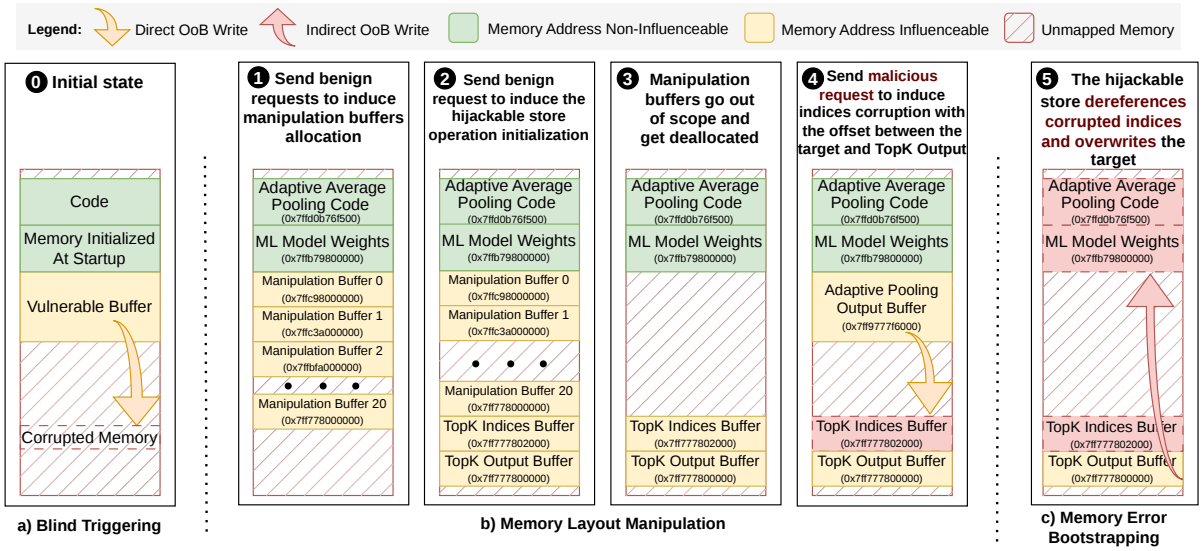


Figure 4: Flow and memory layout during various stages of our attacks against PyTorch inference.

spanning up to $2^{32} - 1$ indices, which corresponds to up to 16 GiB of addressable range above the intended output buffer for 4-byte granularity, and similarly large ranges for other data types. This makes the gatherTopK kernel a powerful bootstrapping gadget, because it amplifies a single OoB write that corrupts its offset buffer into a near-arbitrary write-what-where primitive.

7.3.2 CuDF: escape_strings as a bootstrapping gadget. CuDF's escape_strings kernel exposes a similar hijackable-store pattern via adversary-hijackable offsets loaded from writable device memory. As shown in Listing 6, the kernel computes an output pointer `d_buffer` by adding an offset loaded from the `d_offsets` array to a base pointer `d_chars` (lines 1–2). It then performs a sequence of byte-wise stores through this derived pointer while escaping any dangerous characters, including quote and newline. If an adversary corrupts the `d_offsets` buffer via a preceding direct OoB write, escape_strings will subsequently dereference adversary-controlled offsets and redirect the destination of these stores. Although each store writes only a single byte, the loop structure and inherent parallelism enable multiple consecutive writes to adversary-chosen addresses, yielding a flexible write-what-where primitive whose effective reach is determined by the corrupted offset range.

7.3.3 Expressivity. Gadgets may restrict the addressable range, e.g., to $2^{32} - 1$ elements above the output buffer's base address in `gatherTopK`. `Escape_strings`'s exact range depends on the template instantiation, as `size_type` can resolve to both 32- and 64-bit signed integers. Gadgets may also constrain the writable values, e.g., `gatherTopK` writes only positive values, and `escape_strings` adds unwanted characters to some strings. While these limitations do not preclude our PoCs (see Section 8), more expressive gadgets exist. For example, CuDF's `concat_strings` allows adversaries to copy arbitrary non-null data.

Takeaway 3

While direct OoB writes are only entry points, combining them with memory layout manipulation and hijackable store gadgets that dereference corruptible addresses can bootstrap these limited overflows into near-arbitrary write-what-where primitives in real GPU software.

8 Proof-of-Concept Exploits

This section evaluates our proposed exploitation techniques on the memory errors from Section 3.1 under the deployment model of Section 4. Our PoCs preserve the deployment model's GPU-side properties relevant to exploitation, including long-lived CUDA context, persistent device-resident state, concurrent streams, and request-driven kernel execution, but model requests within the same process to remove network-induced timing variability. Accordingly, we establish the GPU-side feasibility of these exploit chains but do not measure end-to-end success rates. In such a setting, network jitter and latency are likely to introduce additional challenges that should also be addressed. Unless otherwise noted (Section 8.2.1), we isolate PyTorch operations, such as adaptive pooling and TopK, from ML models to avoid mixing exploitability with synthesizing model inputs that induce specific internal activations, a separate challenge discussed in Section 10.3. We assume that deployments meet any configuration and hardware dependencies (e.g., a large maximum batch size and sufficient VRAM) to trigger and exploit these memory errors. Real deployments may, however, choose other parameters, e.g., to optimize performance or prevent network congestion.

As our techniques are not inherently tied to integer overflows, other on-device memory errors that yield comparable primitives such as adjacent OoB writes may also facilitate exploitation. However, we leave experimental validation across other root-cause classes to future work.

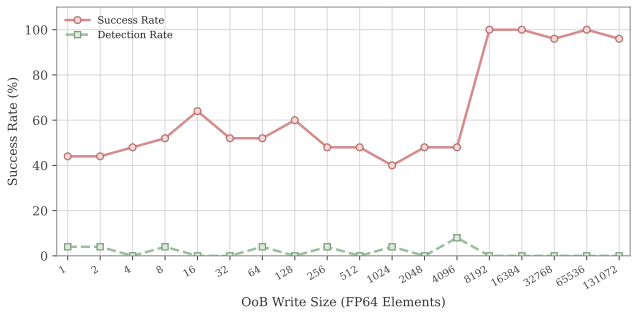


Figure 5: Success and detection rates of indirect model corruption as a function of corruption size with 25 runs per size.

8.1 DoS

As discussed in Section 3.2, blindly triggering device-side memory errors typically results in illegal memory accesses to unmapped GPU addresses. Such faults terminate the CUDA context, rendering the application unavailable for subsequent requests until a monitoring system restarts it. As each restart reinitializes the vulnerable state, adversaries can repeatedly crash the restarted instance, triggering persistent DoS and preventing legitimate use.

We find DoS readily achievable through Section 3.1 errors in PyTorch's adaptive pooling and CuDF's fused concatenation kernels. In both cases, the attack succeeds even when executed in isolation, without relying on specific memory layouts or concurrent activity, making it independent of adversarial knowledge (i.e., **S1-S3**).

8.2 ML Model Degradation and Manipulation

Using the bootstrapping techniques introduced in Section 7, adversaries can coerce OoB writes to overlap with model weights, thereby corrupting the ML model parameters. This produces an integrity violation, in which the victim model continues to respond to inference queries but produces degraded or attacker-influenced outputs. As this attack raises no errors, degradation persists until manually detected and repaired.

Beyond random degradation, which is independent of adversarial knowledge (i.e., possible under **S1-S3**), adversaries can manipulate specific weights identified and retrained offline to steer the model's output. Unless paired with a weight leakage primitive, such controlled manipulation requires weight knowledge (**S1**).

8.2.1 Direct Model Influence. Attackers may overlap *direct* OoB writes with model parameters, e.g., if they can coerce buffers vulnerable to integer-overflow-induced memory errors above these parameters. In our evaluation, corrupting only 0.41% of the weights of PyTorch's Wide ResNet-101 using *direct* OoB writes triggered in a co-resident vulnerable model (VGG11) reduces accuracy to a random guess (0.10%). In our controlled, isolated test, these attacks achieved a 100% success rate, showing that repeated layout-manipulation sequences can produce the same (relative) memory layout on each iteration.

8.2.2 Indirect Model Influence. In many cases, the victim will initialize ML model parameters at startup, placing them at the top of the address space, out of reach of *direct* OoB writes. We address this by bootstrapping the OoB write from PyTorch's adaptive pooling layers using the `gatherTopK` gadget to create an *indirect* OoB write to an address of our choosing. Because the indices used by `gatherTopK` are short-lived, our attack depends on creating an inter-stream data race using two attacker clients that issue concurrent requests, routed to separate host threads and CUDA streams.

Figures 4 and 6 illustrate the memory layout and attack flow, respectively. The *target client* continuously issues benign inference API requests that induce the victim to execute `gatherTopK`, while the *vulnerable client* issues requests that require the victim to execute adaptive pooling. Note that both clients may request inference of the same model if the corresponding request handler executes both operations. Through a series of valid API requests, these clients manipulate the victim's GPU memory layout to position the `TopK` indices buffer (`0x7ff777802000`) and the adaptive pooling output (`0x7ff9777f6000`) buffer, such that the OoB write from adaptive pooling overlaps with this indices buffer. This erroneous store operation overwrites the `TopK` layer's offsets with the offset between the TopK output buffer (`0x7ff777800000`) and the target model weights (`0x7ffb79800000`), i.e., $\frac{\&target - \&topk\_output}{topk\_element\_size}$.

The interleaving of various parts of the TopK and adaptive pooling operations yields three possible outcomes. (*i*) The attack succeeds if adaptive pooling and the corresponding direct OoB write occur between the initialization and the use of the `TopK` offset buffer. This yields a write-what-where primitive, where the `gatherTopK` kernel overwrites selected weights of the victim model. (*ii*) The attack bails out cleanly because the corruption occurs after the `TopK` layer finishes, but before a subsequent initialization. (*iii*) The corruption occurs during the execution of `gatherTopK`, resulting in a data race and partial corruption that usually leads to illegal accesses and context termination. In both cases, adversaries can retry after the system returns to a consistent state.

We quantify practicality by adjusting the number of corrupted elements and running each size 25 times (see Figure 5). All corruption sizes degrade the victim ResNet-18 model's accuracy to a random guess ($\sim 0.10-0.30\%$). Across configurations, attack success primarily depends on thread interleaving and the allocation-reuse behavior of PyTorch's caching allocator and Python's garbage collector. For smaller corruption sizes (< 8192), attacks succeed in approximately 40-60% of controlled tests, corresponding to an expected 2-3 attempts until success. For larger corruption sizes (≥ 8192 elements), we consistently achieve a success rate above 95%, showing that, in our controlled in-process setting, we reliably achieve interleaving (*i*), but that small variations in the exact buffer addresses cause only large overwrites to overlap with the weights. Fewer than 2% of all controlled tests produce detectable errors, i.e., crashes due to partial corruption, i.e., interleaving (*iii*). In end-to-end scenarios, network effects like jitter and latency will likely significantly degrade reliability. Yet, given automatic recovery or silent failure, probabilistic success can translate into eventual exploitation.

8.3 Sensitive Data Leakage

Beyond active corruption, adversaries can exploit OoB reads to leak sensitive GPU-resident data. We observe that many memory errors, including the one in CuDF's fused concatenation kernel,

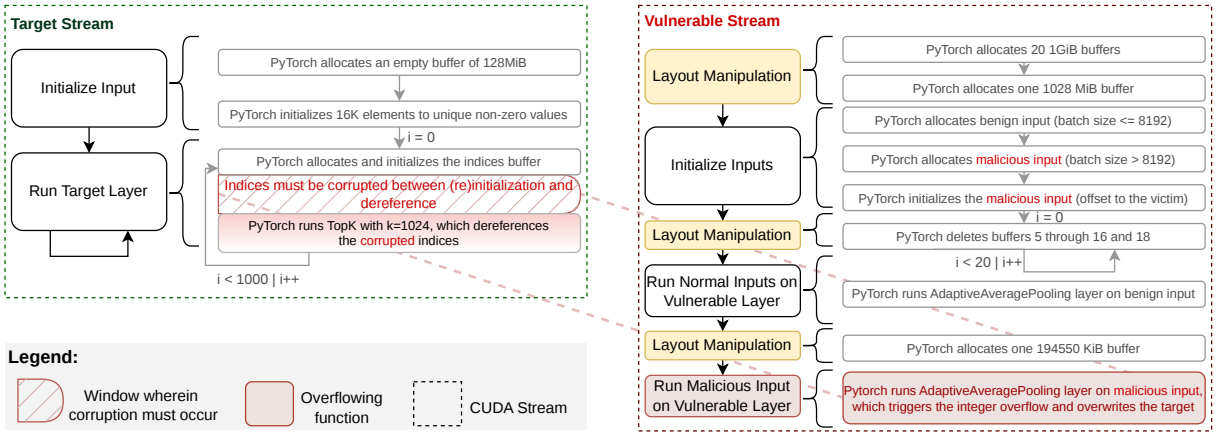


Figure 6: To bootstrap the memory error, the adversary induces framework-level actions on the victim system such as the allocation of massaging buffers and target buffers or the execution of vulnerable kernels and bootstrapping gadgets via valid inference API requests, as they cannot directly allocate memory or call operators.

pair OoB reads and writes. This PoC leverages this read/write pairing in CuDF's fused concatenate (Listing 2) together with layout manipulation to read data OoB from device memory adjacent to the first malicious request and write it OoB into the output of a second concurrent attacker-controlled request. This enables data exfiltration without relying on host-side vulnerabilities. A single pass leaked at most 2^{28} elements.

8.4 Code Injection

Finally, we demonstrate that the lack of an active W⊕X policy allows adversaries to escalate a write-what-where primitive into device-side code injection by overwriting the code region at the top of the global memory address space.

We adapt our indirect-manipulation PoC (Section 8.2.2) to overwrite the adaptive pooling kernel's entry point (`0x7ffd0b76f500`) by changing the offset written by the bootstrapped store operation. Our simplest variant replaces only the first instruction with `EXIT`, causing early termination and leaving the output buffer empty, thereby achieving a *silent* DoS. A second variant modifies this kernel's output buffer, allowing adversaries to influence the model's output without altering the model weights or to leak attacker-selected device memory. A final variant redirects device control flow to an attacker-controlled address relative to the hijacked function. We use this final variant to waste the victim's (computational) power by entering an infinite loop. This attack occupies all SMs, consuming a consistent 132 W on an RTX 5090 until manually terminated, and thereby blocks any subsequently launched kernels.

The first variant inherits the indirect-manipulation PoC's success and detection rates, yielding 48% and 0% across 25 runs, respectively. While the leakage variant had the same success rate as the first variant, all failed attempts were detected because the injected instructions depend on their surroundings, yielding a detection rate of 52%. We fine-tuned the final variant to be independent of its location within the hijacked kernel. It succeeded on all 25 runs despite overwriting only 4 instructions.

Takeaway 4

Once bootstrapped into expressive write primitives, device-side memory errors enable concrete exploitation goals, such as silent model manipulation, data leakage, and device-side code injection. The enduring lesson from the CPU domain is that, while memory errors persist and exploitation techniques evolve, foundational defenses such as W⊕X remain essential for raising the bar against memory corruption.

9 Mitigations

Modern GPUs implement only a limited subset of CPU memory-safety defenses. The mitigations available in NVIDIA's current toolchain, ASLR and stack canaries, do not prevent our PoCs.

Consistent with prior work [86, 107], we find that GPU ASLR adds the same random offset to the base address of most global memory regions, preserving their relative layout. Although absolute addresses vary across executions, our exploits avoid absolute addressing and instead rely solely on relative addressing. In short, NVIDIA's current ASLR implementation does *not* prevent our PoCs.

Stack canaries provide similarly limited protection. They only protect return addresses stored in local memory in functions that NVCC classifies as "high-risk" and are disabled by default. Even when enabled, they are ineffective in practice: return addresses and other sensitive values are rarely spilled to local memory, and targeted overwrites achieved via memory-error bootstrapping can avoid corrupting canary values. As we do not corrupt local memory or return addresses, stack canaries do not prevent our exploits.

Modern GPUs still lack a W⊕X policy by default [52, 83], a crucial defense against code-injection attacks. Ramponi et al. [83] demonstrated that enforcing W⊕Xon commodity GPUs is feasible at negligible overhead and eliminates the most immediate device-side control-flow-hijacking vectors, e.g., code injection.

Although W⊕X does not prevent data-corruption attacks, including our ML model degradation and manipulation exploits, numerous debugging-oriented sanitizers and runtime mitigations have been proposed [9, 11, 19, 35, 36, 42, 51, 63, 64, 66, 76, 95, 98, 99, 103, 104]. However, these tools have seen limited adoption due to deployment

complexity, hardware requirements, incomplete coverage, or high performance overhead. Even projects that employ sanitizers during testing continue to ship memory errors [1, 21], as triggering erroneous paths often requires specific inputs.

Due to the performance-critical nature of GPU code, we identify host input and shape validation (i.e., ensuring no kernel is launched with bounds that it cannot handle) as a more practical mitigation. However, such a mitigation requires a priori knowledge of where memory errors may manifest in kernels.

Given the scale and complexity of modern GPU software, we argue that static application security testing (SAST) is a necessary complement to runtime defenses. While popular SAST tools, such as Semgrep [89] and CodeQL [48], do not natively support CUDA C++, their C/C++ backends could be extended, and several commercial tools have recently introduced CUDA support [12, 37, 81]. Rules tailored to GPU memory errors, such as missing thread-index checks and integer-overflow patterns, could proactively eliminate entire classes of vulnerabilities and prevent multi-year patch cycles caused by repeated instances of the same error [71].

10 Discussion

10.1 Broader Exploitation Goals

Our PoCs demonstrate only a subset of the outcomes enabled by device-side memory errors. More generally, the techniques in Sections 7–8 transform constrained OoB writes into more expressive primitives, most notably write-what-where capabilities and, in the absence of W⊕X, device-side code injection.

These primitives provide a new entry point to a broad class of adversarial goals that prior work has achieved through other GPU attack vectors, such as side channels that typically rely on remote code execution as a precursor. In particular, memory corruption with the potential for device-side code execution can enable exploitation scenarios including cryptojacking [97], password-cracking botnets [7], and resource-wasting attacks [90]. It can also facilitate cross-process data exfiltration from device memory, as demonstrated by several side-channel exploits [39, 57, 58, 65, 75, 77, 91, 102, 106]. Realizing these end goals in full requires additional, non-trivial engineering efforts, which we leave to future work.

10.2 Other Platforms

We developed and evaluated all our PoC exploits on an NVIDIA RTX 5090. We confirmed that our DoS, model degradation, and targeted model-manipulation PoCs required only minor modifications, e.g., due to slight variations in the memory layout or timing, to function correctly across three other NVIDIA architectures, using NVIDIA V100, A100, and H100 GPUs, thereby verifying the feasibility of our techniques on these generations. Porting code injection requires more manual effort and is highly complex due to NVIDIA's unstable, undocumented, architecture-specific SASS encoding. While we confirmed that device code remains writable during kernel execution on all tested architectures, a precursor for code injection, we leave proving the generalizability of our techniques via architecture-tailored or architecture-agnostic (i.e., consisting only of instructions with constant encodings across the targeted architectures) payloads to future work.

While this paper focuses on NVIDIA CUDA C++ applications, memory errors also arise across other accelerator ecosystems. AMD ROCm's HIP C++ [8, 56] is intentionally close to CUDA. Many projects, including PyTorch, share kernel code via compiler macros, resulting in the same memory errors across both platforms. We verified that the same PyTorch orchestration code and inputs reproduce the adaptive pooling errors on ROCm devices, causing the GPU context to terminate and resulting in DoS. Our other PoCs do not reproduce on these devices due to slight differences in the allocation behavior. We leave further analysis and full PoC development to future work.

Other accelerator programming frameworks remain C/C++ dialects, including OpenCL [50], Huawei AscendCL [55], and Intel OneAPI [16]. Even Pythonic accelerator frameworks, such as Triton [100], do not inherit Python's memory safety guarantees. Despite their high-level syntax, they ultimately provide direct, pointer-like control over device memory and remain susceptible to traditional spatial memory errors. We encourage future work to investigate memory safety on these platforms.

10.3 Input Synthesis for ML Inference

While our DoS, ML model degradation, and data-leakage exploits relied solely on input *size* to trigger memory errors and shape the memory layout, both targeted model manipulation and code injection additionally required inputs that induce specific activations. Adaptive pooling does not occur first in any vulnerable model. Instead, CPU-side pre-processing including scaling, normalization, and tokenization, and layers such as convolution, ReLU, and pooling, precede it. While crafting activations for the vulnerable layer itself is relatively straightforward, generating inputs that induce such activations depends on the model weights. As a result, payload-dependent attacks become infeasible under **S3** unless adversaries leak the necessary weights. Even when adversaries can fully reproduce the victim's setup, including the model weights, and all transformations are deterministic (i.e., no randomized pre-processing), synthesizing inputs that induce precise activations at a particular layer remains challenging. Constraint-solving approaches, e.g., using mixed-integer linear programming, quickly become impractical due to the high dimensionality, large number of constraints, and non-linear operations involved.

To address this challenge, we developed a gradient-based input synthesis technique that optimizes the inputs to match a target activation pattern in the vulnerable layer. Starting from random noise, we iteratively update the input using gradient descent while keeping all network parameters fixed. At each iteration, PyTorch's Adam optimizer minimizes the loss function that measures the deviation between the target and the observed activations. Conceptually, this approach resembles white-box fuzzing techniques that guide input generation using internal program state.

Our preliminary experiments yielded encouraging results: the synthesized activations matched the target values except for the lower three bits. While such deviations may introduce minor side effects, memory-error bootstrapping remains feasible with adapted payloads, and tuning optimization parameters (e.g., the learning rate, loss function, or number of iterations) may reduce them.

11 Related Work

Prior GPU exploitation research primarily attempted to port CPU memory-error attacks, such as code injection and code reuse, to GPUs [34, 52, 67, 73, 86]. These works demonstrate that artificially introduced spatial memory errors that affect global and local memory can corrupt function pointers, `vtable` pointers, or spilled return addresses, enabling control-flow hijacking via classical techniques. While their findings are cause for concern, our analysis of real-world GPU applications reveals that these data structures are exceedingly rare in writable device memory and that return addresses are rarely spilled to local memory, which limits the feasibility of prior attack strategies. We show that real GPU memory errors can be triggered under certain deployment conditions (e.g., large maximum batch sizes and sufficient VRAM) and that expressive exploitation requires additional steps not addressed by prior work, i.e., bootstrapping constrained OoB writes into write-what-where primitives.

Our taint analysis resembles Einstein [59], where Johannesmeyer et al. use taint tracking to synthesize data-only exploits on CPUs by identifying adversary-influenced values that flow into system calls. While similar in spirit, Einstein operates on CPU binaries, assumes their gadgets are present in writable memory, and focuses on automatic exploit generation. Our framework aims to aid in reverse-engineering GPU code and identifying bootstrapping gadgets. Therefore, it targets CUDA code, where memory spaces differ substantially and open-source data-flow analysis frameworks are limited. Hayes et al. [53] presented the only other dynamic taint analysis for GPUs. Their runtime defense prevents data exfiltration from device memory and detects GPU malware. It relies on a custom SASS instrumentation engine that is incompatible with modern NVIDIA architectures, limiting applicability. While this instrumentation engine is open-source, the taint analysis is not. In contrast, our tool inherits compatibility from NVBIT [101], a tool developed and maintained by NVIDIA, which has been compatible with every NVIDIA GPU architecture since its inception, making it the first open-source dynamic taint analysis for NVIDIA Ampere, Hopper, Ada Lovelace, and Blackwell GPUs.

12 Conclusion

This paper challenges the common belief that GPU memory errors pose mere reliability issues by demonstrating that adversaries can trigger device-side memory errors in real-world device code remotely under specific deployment conditions. Our controlled evaluation shows that adversaries can corrupt sensitive GPU-resident data, and even bootstrap limited-range *direct* memory errors into near-arbitrary write-what-where primitives, despite current GPU memory safety mitigations such as ASLR and stack canaries. Such capabilities can enable denial-of-service, model degradation, targeted model manipulation, information leakage, and ultimately device-side code execution.

To develop these exploits, we analyzed 116 memory errors across 19 codebases and built a custom taint-tracking framework that identifies adversary-controllable data flows, characterizes memory-error behavior, and reveals bootstrapping gadgets. Our findings highlight that, as GPUs continue to underpin ML and an increasing number of other domains, the need for GPU vendors and developers to adopt stronger memory-safety defenses, most notably W⊕X, SAST, and bounds checking, grows. More broadly, treating memory errors in accelerator code as security vulnerabilities is essential, as these devices take on a central role in modern computing.

Acknowledgements

We would like to thank the CCS reviewers and, in particular, the shepherd of this paper for their feedback. Furthermore, we thank all colleagues from DistriNet and KU Leuven whose insightful discussions and feedback have helped shape this research.

This research is partially funded by the Internal Funds KU Leuven, and by the Cybersecurity Research Program Flanders.

References

[1] RAPIDS AI. 2019. cuDF - GPU DataFrames - [FEA] gpuCI should run tests under cuda-memcheck #904. https://github.com/rapidsai/cudf/issues/904.
[2] RAPIDS AI. 2020. cuDF - GPU DataFrames. https://github.com/rapidsai/cudf.
[3] RAPIDS AI. 2022. BlazingSQL. https://github.com/rapidsai/blazingsql-release-staging.
[4] RAPIDS AI. 2025. cuML - RAPIDS Machine Learning Library. https://github.com/rapidsai/cuml.
[5] RAPIDS AI. 2025. cuVS - a library for vector search and clustering on the GPU. https://github.com/rapidsai/cuvs.
[6] RAPIDS AI. 2025. RAFT: Reusable Accelerated Functions and Tools for Vector Search and More. https://github.com/rapidsai/raft.
[7] Ibrahim Alkhwaja, Mohammed Albugami, Ali Alkhwaja, Mohammed Alghamdi, Hussam Abahussain, Faisal Alfawaz, Abdullah Almurayh, and Nasro Min-Allah. 2023. Password Cracking with Brute Force Algorithm and Dictionary Attack Using Parallel Programming. *Applied Sciences* 13, 10 (2023). doi:10.3390/app13105979
[8] AMD. 2016. HIP: Heterogeneous-Compute Interface for Portability. https://github.com/ROCm/HIP.
[9] AMD. 2024. Using the AddressSanitizer on a GPU (beta release). https://rocm.docs.amd.com/en/docs-6.0.0/conceptual/using-gpu-sanitizer.html.
[10] ArrayFire. 2025. ArrayFire - The Fastest Library for GPUs. https://arrayfire.com/.
[11] Thomas M. Baumann and José Gracia. 2014. Cudagrind: Memory-Usage Checking for CUDA. In *Tools for High Performance Computing 2013*. Springer International Publishing, Cham, 67–78. https://arxiv.org/pdf/1310.0901
[12] Ricardo Camacho. 2025. Bringing Rigorous Static Analysis to Your CUDA Code. https://www.parasoft.com/blog/rigorous-static-analysis-cuda-code/.
[13] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *IEEE IISWC*. doi:10.1109/IISWC.2009.5306797
[14] Catalin Cimpanu. 2019. Microsoft: 70 percent of all security bugs are memory safety issues. https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/.
[15] Clamchowder. 2025. Blackwell: Nvidia's Massive GPU. https://old.chipsandcheese.com/2025/06/28/blackwell-nvidias-massive-gpu/.
[16] Intel Co. 2024. oneAPI: A New Era of Heterogeneous Computing. https://www.intel.com/content/www/us/en/developer/tools/oneapi/overview.html.
[17] Peter Collingbourne, Cristian Cadar, and Paul H. J. Kelly. 2012. Symbolic Testing of OpenCL Code. In *Hardware and Software: Verification and Testing*. Springer Berlin Heidelberg, 203–218. doi:10.1007/978-3-642-34188-5_18
[18] NVIDIA Corporation. 2009. CUDA Core Compute Libraries (CCCL). https://github.com/NVIDIA/cccl.
[19] NVIDIA Corporation. 2013. CUDA-MEMCHECK. https://developer.nvidia.com/cuda-memcheck
[20] NVIDIA Corporation. 2021. RAPIDS Accelerator for Apache Spark. https://docs.nvidia.com/spark-rapids/index.html.
[21] NVIDIA Corporation. 2023. RAPIDS Accelerator for Apache Spark - Run compute-sanitizer memcheck on tests #1221. https://github.com/NVIDIA/spark-rapids-jni/issues/1221.
[22] NVIDIA Corporation. 2024. CUDA C++ Programming Interface. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-interface.
[23] NVIDIA Corporation. 2024. NVIDIA Blackwell Architecture. https://www.nvidia.com/en-us/data-center/technologies/blackwell-architecture/.
[24] NVIDIA Corporation. 2024. NVIDIA CUB. https://github.com/NVIDIA/cub.
[25] NVIDIA Corporation. 2026. CUDA C++ Best Practices Guide. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/.
[26] NVIDIA Corporation. 2026. NVIDIA Triton Inference Server. https://docs.nvidia.com/deeplearning/triton-inference-server/user-guide/docs/index.html.

[27] NVIDIA Corporation. 2026. nvJPEG Libraries: GPU-accelerated JPEG decoder, encoder and transcoder. https://developer.nvidia.com/nvjpeg.

[28] CuDF. 2025. Fused Concatenate. https://github.com/rapidsai/cudf/blob/main/cpp/src/copying/concatenate.cu#L185C18-L185C42.

[29] CuDF. 2026. Escape Strings. https://github.com/rapidsai/cudf/blob/main/cpp/src/io/csv/writer_impl.cu#L67.

[30] Bill Dally. 2023. Hardware for Deep Learning. *IEEE HCS* (29 Aug. 2023). https://hc2023.hotchips.org/assets/program/conference/day2/Keynote%202/Keynote-NVIDIA_Hardware-for-Deep-Learning.pdf

[31] Dorothy E. Denning and Peter J. Denning. 1977. Certification of programs for secure information flow. *Commun. ACM* 20, 7 (July 1977), 504–513. doi:10.1145/359636.359712

[32] PaddlePaddle Developers. 2025. Paddle - PArallel Distributed Deep LEarning: Machine Learning Framework from Industrial Practice. https://github.com/PaddlePaddle/Paddle.

[33] TensorFlow Developers. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/

[34] Bang Di, Jianhua Sun, and Hao Chen. 2016. A study of overflow vulnerabilities on GPUs. In *IFIP NPC*. Springer, 103–115. doi:10.1007/978-3-319-47099-3_9

[35] Bang Di, Jianhua Sun, Hao Chen, and Dong Li. 2021. Efficient Buffer Overflow Detection on GPU. *IEEE Trans. Parallel Distrib. Syst.* 32, 5 (May 2021), 1161–1177. doi:10.1109/TPDS.2020.3042965

[36] Bang Di, Jianhua Sun, Dong Li, Hao Chen, and Zhe Quan. 2018. GMOD: a dynamic GPU memory overflow detector. In *ACM PACT*. doi:10.1145/3243176.3243194

[37] Black Duck. 2025. Coverity - Static Analysis Language and Framework Support. https://www.blackduck.com/static-analysis-tools-sast/languages-and-frameworks.html.

[38] Gregory J. Duck and Roland H. C. Yap. 2016. Heap Bounds Protection with Low Fat Pointers. In *ACM CC*. doi:10.1145/2892208.2892212

[39] Sankha Baran Dutta, Hoda Naghibijouybari, Arjun Gupta, Nael Abu-Ghazaleh, Andres Marquez, and Kevin Barker. 2023. Spy in the GPU-box: Covert and Side Channel Attacks on Multi-GPU Systems. In *ACM ISCA*. doi:10.1145/3579371.3589080

[40] Peter Eastman and Vijay Pande. 2010. OpenMM: A Hardware-Independent Framework for Molecular Simulations. *Computing in Science & Engineering* 12, 4 (2010), 34–39. doi:10.1109/MCSE.2010.27

[41] Marwa Elteir, Heshan Lin, Wu-chun Feng, and Tom Scogland. 2011. StreamMR: An Optimized MapReduce Framework for AMD GPUs. In *2011 IEEE 17th International Conference on Parallel and Distributed Systems*. 364–371. doi:10.1109/ICPADS.2011.131

[42] Christopher Erb, Mike Collins, and Joseph L. Greathouse. 2017. Dynamic buffer overflow detection for GPGPUs. In *IEEE/ACM CGO*. doi:10.1109/CGO.2017.7863729

[43] Chris Evans. 2014. Exploiting CVE-2014-0556 in Flash.

[44] Hugging Face. 2018. Models - Transformers. https://huggingface.co/docs/transformers/index.

[45] Hugging Face. 2025. Models - Hugging Face. https://huggingface.co/models.

[46] Johannes Gaessler. 2024. GGML.org Llama.cpp mul_mat_vec. https://github.com/JohannesGaessler/llama.cpp/blob/6768787dd36e2f481fa24282e9246367704e573d/ggml/src/ggml-cuda/mmv.cu#L5.

[47] ggml org. 2025. LLAMA.cpp - LLM inference in C/C++. https://github.com/ggml-org/llama.cpp.

[48] GitHub, Inc. 2024. CodeQL: Semantic Code Analysis Engine. https://codeql.github.com.

[49] Floris Gorter and Cristiano Giuffrida. 2025. RangeSanitizer: Detecting Memory Errors with Efficient Range Checks. In *USENIX Security*. https://www.usenix.org/system/files/usenixsecurity25-gorter.pdf

[50] Khronos Group. 2013. OpenCL - The Open Standard for Parallel Programming of Heterogeneous Systems. https://www.khronos.org/opencl/ Section: API.

[51] Khronos Group. 2014. Webcl-Validator. https://github.com/KhronosGroup/webcl-validator.

[52] Yanan Guo, Zhenkai Zhang, and Jun Yang. 2024. GPU Memory Exploitation for Fun and Profit. In *USENIX Security*. https://www.usenix.org/conference/usenixsecurity24/presentation/guo-yanan

[53] Ari B. Hayes, Lingda Li, Mohammad Hedayati, Jiahuan He, Eddy Z. Zhang, and Kai Shen. 2017. GPU Taint Tracking. In *USENIX ATC*. https://www.usenix.org/conference/atc17/technical-sessions/presentation/hayes

[54] Sean Heelan, Tom Melham, and Daniel Kroening. 2018. Automatic heap layout manipulation for exploitation. In *USENIX security*.

[55] Huawei. 2022. Ascend Computing Language (AscendCL). https://www.hiascend.com.

[56] Advanced Micro Devices Inc. 2024. AMD ROCm™ Software. https://rocm.docs.amd.com/projects/HIP/en/latest/reference/cpp_language_extensions.html.

[57] Zhen Hang Jiang, Yunsi Fei, and David Kaeli. 2016. A complete key recovery timing attack on a GPU. In *IEEE HPCA*. https://doi.org/10.1109/HPCA.2016.7446081

[58] Zhen Hang Jiang, Yunsi Fei, and David Kaeli. 2017. A Novel Side-Channel Timing Attack on GPUs. In *ACM GLVLSI*. doi:10.1145/3060403.3060462

[59] Brian Johannesmeyer, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. 2024. Practical Data-Only Attack Generation. In *USENIX Security*. https://www.usenix.org/conference/usenixsecurity24/presentation/johannesmeyer

[60] jwnhy. 2025. [CUDA] Illegal Memory Access with AdaptiveAvgPool2d #145349. https://github.com/pytorch/pytorch/issues/145349.

[61] jwnhy. 2025. [CUDA] Illegal Memory Access with AdaptiveMaxPool2d #145453. https://github.com/pytorch/pytorch/issues/145453.

[62] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. arXiv:2309.06180 [cs.LG] https://arxiv.org/abs/2309.06180

[63] 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 *IEEE HPCA*. doi:10.1109/HPCA61900.2025.00122

[64] Jaewon Lee, Yonghae Kim, Jiashen Cao, Euna Kim, Jaekyu Lee, and Hyesoon Kim. 2022. Securing gpu via region-based bounds checking. In *ISCA*. doi:10.1145/3470496.3527420

[65] Sangho Lee, Youngsok Kim, Jangwoo Kim, and Jong Kim. 2014. Stealing Webpages Rendered on Your Browser by Exploiting GPU Vulnerabilities. In *IEEE S&P*. 19–33. doi:10.1109/SP.2014.9

[66] Hongyi Lu, Fengwei Zhang, Zhenkai Zhang, Shuai Wang, and Yanan Guo. 2026. CUSAFE: Capturing Memory Corruption on NVIDIA GPUs. In *USENIX Security*. https://www.usenix.org/conference/usenixsecurity26/presentation/lu

[67] Andrea Miele. 2016. Buffer overflow vulnerabilities in CUDA: a preliminary analysis. *Journal of Computer Virology and Hacking Techniques* 12 (2016), 113–120. https://arxiv.org/abs/1506.08546

[68] MITRE. 2025. CVE-2025-23274. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2025-23274.

[69] MITRE. 2026. CVE-2026-53923. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2026-53923.

[70] Yeoul Na. 2023. -fbounds-safety. Enforcing bounds safety for production C code. *EuroLLVM Developers' Meeting* (11 May 2023). https://llvm.org/devmtg/2023-05/slides/TechnicalTalks-May11/01-Na-fbounds-safety.pdf

[71] nvdbaranec. 2022. cuDF - GPU DataFrames - Prevent grid stride loop overflow in libcudf kernels #10368. https://github.com/rapidsai/cudf/issues/10368.

[72] OpenMM. 2020. findBlocksWithInteractions CUDA Kernel. https://github.com/peastman/openmm/blob/20501a0979d24bbd84d51e16f53d017144814e19/platforms/cuda/src/kernels/findInteractingBlocks.cu#L178C28-L178C54.

[73] 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. doi:10.1016/j.cose.2020.102115

[74] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *NeurIPS*. doi:10.5555/3454287.3455008

[75] Roberto Di Pietro, Flavio Lombardi, and Antonio Villani. 2016. CUDA Leaks: A Detailed Hack for CUDA and a (Partial) Fix. *ACM Trans. Embed. Comput. Syst.* 15, 1, Article 15 (Jan. 2016), 25 pages. doi:10.1145/2801153

[76] James Price and Simon McIntosh-Smith. 2015. Oclgrind: an extensible OpenCL device simulator. In *IWOCL*. doi:10.1145/2791321.2791333

[77] Frederik Dermot Pustelnik, Xhani Marvin Saß, and Jean-Pierre Seifert. 2024. Whispering Pixels: Exploiting Uninitialized Register Accesses in Modern GPUs. arXiv:2401.08881 [cs.CR]

[78] PyTorch. 2017. TorchVision - Datasets, Transforms and Models specific to Computer Vision. https://github.com/pytorch/vision.

[79] PyTorch. 2023. [CUDA] 2D Adaptive Max Pooling. https://github.com/pytorch/pytorch/blob/main/aten/src/ATen/native/cuda/AdaptiveMaxPooling2d.cu#L46.

[80] PyTorch. 2026. [CUDA] Tensor TopK. https://github.com/pytorch/pytorch/blob/main/aten/src/ATen/native/cuda/TensorTopK.cu#L42.

[81] QT. 2025. Mastering CUDA Code Quality: Your Playbook for Developing Game-Changing CUDA Applications. https://www.qt.io/hubfs/Axivion_documents_EN/Axivion_for_CUDA_Mastering_Software_Quality.pdf.

[82] Rajat Raina, Anand Madhavan, and Andrew Y. Ng. 2009. Large-scale deep unsupervised learning using graphics processors. In *ACM ICML*. doi:10.1145/1553374.1553486

[83] Carlo Ramponi, Adriaan Jacobs, Luigi Dell'Eva, Stijn Volckaert, Silviu Vlasceanu, Mahmoud Ammar, and Bruno Crispo. 2026. Mitigating Code Injection Attacks on Modern GPUs. In *Proceedings of the 33rd ACM Conference on Computer and Communications Security (CCS'26)*.

[84] Ray. 2026. Ray Serve: Scalable and Programmable Serving. https://docs.ray.io/en/latest/serve/index.html.
[85] Alex Rebert, Chandler Carruth, Jen Engel, and Andy Qin. 2024. Safer with Google: Advancing Memory Safety. https://security.googleblog.com/2024/10/safer-with-google-advancing-memory.html
[86] Jonas Roels, Adriaan Jacobs, and Stijn Volckaert. 2025. CUDA, Woulda, Shoulda: Returning Exploits in a SASS-y World. In *EuroSec*. doi:10.1145/3722041.3723099
[87] Sihyun Roh, Woohyuk Choi, Jaeyoung Chung, Yoochan Lee, Suhwan Song, and Byoungyoung Lee. 2026. GHost in the SHELL: A GPU-to-Host Memory Attack and Its Mitigation. In *IEEE S&P*. doi:10.1109/SP63933.2026.00047
[88] Google Cloud Security. 2025. Mandiant M-Trends 2025 Report. https://services.google.com/fh/files/misc/m-trends-2025-en.pdf.
[89] Semgrep, Inc. 2024. Semgrep: Fast, Open-Source Static Analysis. https://semgrep.dev.
[90] Ilia Shumailov, Yiren Zhao, Daniel Bates, Nicolas Papernot, Robert Mullins, and Ross Anderson. 2021. Sponge Examples: Energy-Latency Attacks on Neural Networks. In *IEEE EuroS&P*. doi:10.1109/EuroSP51992.2021.00024
[91] Tyler Sorenson and Heidy Khlaaf. 2024. LeftoverLocals: Listening to LLM responses through leaked GPU local memory. Blog. https://blog.trailofbits.com/2024/01/16/leftoverlocals-listening-to-llm-responses-through-leaked-gpu-local-memory/
[92] Alexander Sotirov. 2007. Heap feng shui in javascript. *Black Hat* (2007). https://www.blackhat.com/presentations/bh-usa-07/Sotirov/Whitepaper/bh-usa-07-sotirov-WP.pdf
[93] Aviral Srivastava. 2026. *221 Silent Integer Truncations in vLLM: How a Single Malicious Model File Can Corrupt GPU Memory in the Most Widely Deployed LLM Inference Engine*. Technical Report. https://github.com/Aviral2642/vllm-integer-truncation-audit Independent security research report.
[94] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. 2012. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing* 127, 7.2 (2012). http://impact.crhc.illinois.edu/Shared/Docs/impact-12-01.parboil.pdf
[95] 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 *ACM ISCA*. doi:10.1145/3579371.3589102
[96] Yifan Sun, Xiang Gong, 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. In *IEEE IISWC*. doi:10.1109/IISWC.2016.7581262
[97] Dmitry Tanana. 2024. Behavior-Based Detection of GPU Cryptojacking. arXiv:2408.14554 [cs.CR] https://arxiv.org/abs/2408.14554
[98] 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. *ACM PLDI* (2023). doi:10.1145/3591225
[99] Mohamed Tarek Ibn Ziad, Sana Damani, Mark Stephenson, Stephen W. Keckler, and Aamer Jaleel. 2026. GPUArmor: A Hardware-Software Co-Design for Efficient and Scalable Memory Safety on GPUs. *ACM TACO* (2026). doi:10.1145/3815783
[100] Philippe Tillet, H. T. Kung, and David Cox. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In *ACM MAPL*. doi:10.1145/3315508.3329973
[101] Oreste Villa, Mark Stephenson, David Nellans, and Stephen W. Keckler. 2019. NVBit: A Dynamic Binary Instrumentation Framework for NVIDIA GPUs. In *IEEE/ACM MICRO*. doi:10.1145/3352460.3358307
[102] 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 *IEEE S&P*. doi:10.1109/SP54263.2024.00084
[103] Jiang Wu, Bang Di, Jianhua Sun, Hao Chen, Xionghu Zhong, DaoKun Hu, and Chenlin Huang. 2019. A Fast and Secure GPU Memory Allocator. In *IEEE HPCC/SmartCity/DSS*. doi:10.1109/HPCC/SmartCity/DSS.2019.00035
[104] Chaochao Zhang and Rui Hou. 2022. LAK: A Low-Overhead Lock-and-Key Based Schema for GPU Memory Safety. In *IEEE ICCD*. doi:10.1109/ICCD56317.2022.00108
[105] Yuhao Zhou, Peng Jia, Jiayong Liu, and Ximing Fan. 2026. Fuzz4Cuda: Fuzzing your NVIDIA GPU libraries through debug interface. *Computers & Security* 161 (2026), 104754. doi:10.1016/j.cose.2025.104754
[106] 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:1605.06610 [cs.CR] https://arxiv.org/abs/1605.06610
[107] Ruofan Zhu, Ganhao Chen, Wenbo Shen, Lyuye Zhang, Dakun Shen, Rui Chang, and Yanan Guo. 2026. Demystifying and Exploiting ASLR on NVIDIA GPUs. In *IEEE S&P*. doi:10.1109/SP63933.2026.00078

Open Science

We provide all artifacts necessary to reproduce the experiments presented in this paper, including:

- the source code of our dynamic taint-analysis framework for NVIDIA GPUs,
- scripts to run our proof-of-concept exploits against PyTorch and CuDF,
- raw evaluation results, scripts to parse them, and an additional section discussing them,
- scripts and documentation to replicate the evaluation environment for taint tracking and exploitation,
- the GPU memory error dataset.

Our artifact is available at: https://github.com/RoelsJonas/From-Prompt-to-Pwn and https://doi.org/10.5281/zenodo.22671972.

Unless otherwise indicated, we evaluated on an NVIDIA RTX 5090 using CUDA Toolkit 13.0 and NVIDIA Driver 580.65.06, running on Ubuntu 24.04 with Linux kernel 6.14.11. Successful exploitation on other systems may require adaptations, such as adjusting offsets due to memory-layout variability or modifying payload encodings for different architectures.

Ethical Considerations

This work primarily aims to raise awareness that memory-safety vulnerabilities in deployed accelerator code are prevalent, potentially exploitable, and that their impact can extend beyond DoS. By demonstrating PoC exploits and attack primitives, we aim to encourage vendors and developers to treat these memory errors as genuine security threats rather than reliability issues.

We conducted all experiments for this paper in replicated, offline, isolated environments of the vulnerable software. At no point during this research did our experiments affect live systems, third-party services, or legitimate users. While we did not exploit any memory errors not yet known to their respective developers, we followed responsible disclosure practices for all vulnerabilities that had not yet been patched.

While our taint-tracking framework assists in understanding data flow and identifying potentially exploitable targets, it does not automate exploit generation. Exploits still require manual reasoning about memory layout, kernel behavior, and input crafting.