GPU-Fuzz: A Constraint-Guided Fuzzer for Finding Memory Errors in GPU-Accelerated Deep Learning Frameworks

NAME

Abstract

GPU memory errors represent an imminent and important threat to the reliability of deep learning (DL) frameworks, leading to system crashes and silent data corruption. However, discovering these bugs is challenging because existing fuzzers for DL systems focus on the neural network structure to find compiler logic errors. While effective for their purpose, they do not systematically explore the intricate operator parameter space where conditions that trigger low-level GPU memory errors lie. This paper introduces GPU-FUZZ, a fuzzer specifically engineered to address this gap. Unlike structure-oriented fuzzers, GPU-FUZZ models the complex relationships between operator parameters as formal constraints. It then uses a constraint solver to generate test cases that probe boundary conditions prone to memory errors in GPU kernels. By applying GPU-FUZZ to major frameworks like PyTorch and TensorFlow, we discovered 14 previously unknown bugs, the majority of which are critical memory-related errors. Our work demonstrates that focusing on the operator parameter space is a highly effective and complementary approach for securing the foundational layers of modern DL systems.

1 Introduction

The security of GPU computations within deep learning (DL) frameworks like PyTorch and TensorFlow is critical. An imminent and important threat in this domain is the prevalence of GPU memory errors, a severe and insidious class of vulnerabilities stemming from the low-level CUDA kernels that implement DL operators. These errors, such as out-of-bounds access or misaligned memory addressing, can lead not only to system crashes but also to silent data corruption, posing a significant threat to the reliability and security of AI applications.

However, discovering these memory-related bugs remains a profound challenge. Existing fuzzers for DL systems are primarily designed to find logical bugs in the compiler stack by generating entire neural networks. This network-level focus, while effective for compiler testing, is ill-suited for uncovering memory errors that lie deep within the parameter space of individual operators.

Our key insight is that uncovering GPU memory errors requires a shift in focus from the network structure to the operator parameters and memory layout. The precise combination of tensor shapes, data types, strides, and other parameters dictates the memory access patterns within a CUDA kernel. To effectively find memory bugs, a fuzzer must be able to reason about these intricate constraints and generate inputs that specifically stress the memory management logic.

To this end, we designed and implemented GPU-FUZZ, a novel fuzzer specifically engineered to address the threat of GPU memory errors. Unlike structure-oriented fuzzers, GPU-FUZZ focuses on the operator level. It translates the complex semantic and memory-related rules of DL operators into formal constraints, which are then solved to generate precise inputs that probe memory-related boundary conditions. This constraint-guided approach enables GPU-FUZZ to bypass the frameworks' frontend validation and directly stress-test the low-level CUDA kernels for memory safety.

The main contributions of this paper are as follows:

- We propose a new fuzzing methodology that targets GPU memory errors by systematically exploring the operator parameter space, a dimension orthogonal to existing network structure fuzzers.
- We design and implement GPU-FUZZ, a system that embodies this methodology by leveraging constraint solving to automatically generate test cases that probe memory-related boundary conditions in low-level CUDA kernels.
- We demonstrate the effectiveness of GPU-FUZZ by discovering 14 previously unknown bugs, the majority being severe memory errors, in major DL frameworks and provide public minimal reproducers to benefit the community.

2 Preliminary and Motivation

2.1 Background

GPU kernels, typically written in CUDA, are susceptible to various memory errors. Our work focuses on critical bugs such as illegal memory accesses (out-of-bounds reads/writes), misaligned memory accesses, and race conditions, which can lead to silent data corruption or system crashes.

To detect these low-level errors, we use NVIDIA's `compute-sanitizer` as our primary test oracle. It is a powerful runtime tool that instruments GPU code to check for memory safety violations. When an error is detected, it terminates the application and provides a detailed report, enabling us to identify bugs that do not cause crashes at the API level.

2.2 Motivation

Our research is motivated by a critical gap in existing fuzzing methodologies for deep learning systems.

Compiler-Centric Focus of Existing Fuzzers. State-of-the-art fuzzers for DL frameworks, such as NNSmith, have primarily focused on testing the compiler stack. Their methodology involves generating structurally valid neural networks to uncover logical inconsistencies and wrong-computation bugs that arise during graph-level optimizations. While highly effective for discovering compiler bugs, this network-level perspective is not designed to probe for low-level vulnerabilities within the CUDA kernels of individual operators.

The Operator-Level Blind Spot for Memory Errors. GPU memory errors—such as out-of-bounds access or misaligned addressing—are not typically triggered by the high-level network architecture. Instead, they are instigated by specific, often boundary-value, combinations of an operator's parameters, including tensor shapes, data types, strides, and padding. These parameters directly dictate the memory access patterns within a CUDA kernel. Consequently, fuzzers that operate at the network structure level have a fundamental blind spot: they do not systematically explore the intricate parameter space of individual operators where these critical memory-related bugs reside.

These observations reveal the need for a paradigm shift in fuzzing for GPU security. We require a fuzzer that moves beyond the network structure to focus directly on the operator level. This motivates the design of GPU-Fuzz, a system engineered to systematically explore the operator parameter space to uncover memory safety violations in low-level CUDA kernels.

3 System Design

This section details the architecture of GPU-Fuzz. As illustrated in Figure 1, the system is designed around a data flow that seamlessly connects constraint-based test case generation with cross-framework execution, monitoring, and log-based reproduction.

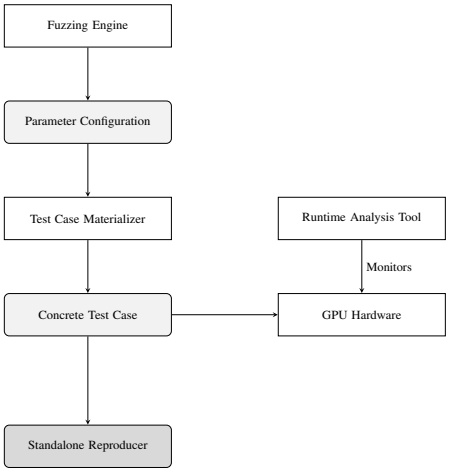

Figure 1: The architecture of the GPU-Fuzz system.

3.1 Constraint Library and Operator Modeling

The foundation of GPU-Fuzz is its ability to generate semantically valid inputs. This is achieved through a Constraint Library that models the operational semantics of deep learning operators. For each target operator (e.g., convolution, pooling, matrix multiplication), we define a set of constraints that capture the legitimate relationships between its parameters, such as tensor shapes, data types, kernel sizes, strides, and padding. These constraints create a framework-agnostic representation of the operator's contract. This approach ensures that any test case generated is, by construction, guaranteed to pass the framework's initial validation checks.

3.2 Constraint-based Test Case Generation

The core of the fuzzing loop resides in the test case generator. The process begins by randomly selecting an operator template from the Constraint Library. The generator then populates the template with randomized, but plausible, target dimensions and parameters. These parameters, along with the operator's formal constraints, are passed to a constraint solver. The solver's task is to find a concrete assignment of values that satisfies all constraints. The result is a complete set of valid parameters (e.g., `input_shape`, `kernel_size`, `stride`) and the corresponding output tensor shape, which together form a framework-agnostic parameter configuration.

For instance, consider a 2D convolution (`Conv2D`). The relationship between input height (H_{in}), output height (H_{out}), padding (P), kernel (K), dilation (D), and stride (S) is governed by the equation: $H_{out} = \lfloor (H_{in} + 2P - D \cdot (K-1) -$

$1)/S\rfloor + 1$. Furthermore, for grouped convolutions, the number of input and output channels must both be divisible by the 'groups' parameter. Our fuzzer translates these rules into a formal constraint system. It might generate a plausible H_{in} and then task the solver with finding a consistent set of values for the remaining parameters that satisfy all interdependent constraints. This ensures the generated parameters are holistically consistent, a prerequisite for bypassing framework validation and testing the underlying kernel.

3.3 Cross-Framework Execution and Monitoring

Once a set of valid parameters is generated, this parameter configuration is materialized into an executable script for a target framework (PyTorch, TensorFlow, or PaddlePaddle). This script creates the necessary tensors, populates them with random data, and invokes the operator on the GPU.

The execution is monitored by NVIDIA's `compute-sanitizer`. This tool can detect a wide range of GPU-level errors, such as out-of-bounds memory accesses and race conditions, that would not typically cause a visible crash at the Python API level. When a violation is detected, `compute-sanitizer` terminates the process and generates a detailed report, which serves as our primary bug oracle.

3.4 Bug Reproducibility

To facilitate the validation and debugging of discovered vulnerabilities, GPU-FUZZ systematically records the conditions that trigger each failure. This record encapsulates the target operator and framework, the specific input parameters (e.g., kernel size, stride), the shapes of all input tensors, and the random seed used for data generation. The raw input tensors are also preserved. This collection of artifacts provides a self-contained and deterministic reproducer for each bug, significantly simplifying the process of verification and root cause analysis for developers.

4 Implementation

Our implementation of GPU-Fuzz consists of approximately 2,000 lines of Python code. The system is orchestrated by a central controller that manages the fuzzing loop, which includes test case generation, execution, and logging.

Constraint Library and Generation. Our constraint library, implemented in `gen/ops.py`, currently provides models for 11 families of operators, including various types of convolutions, pooling, and element-wise arithmetic operations. We prioritized these operator families because they are not only fundamental to a wide range of deep learning models but also feature complex memory access patterns, making them a high-priority target for uncovering memory-related vulnerabilities. Table 1 lists the supported operator families. We use the Z3 theorem prover from Microsoft Research as our backend constraint solver. Building a model for a new operator family, such as `AbsConv`, typically requires 100-150 lines of code to define its parameters as symbolic variables and encode the mathematical relationships between them as formal constraints. The `materialize` method in each operator class then queries the Z3 solver in a loop. After finding a valid solution, it adds a new constraint to exclude the current solution, thereby compelling the solver to find a novel and distinct set of valid parameters in the subsequent iteration. This process is essential for exploring a diverse range of configurations.

Framework Instantiation. Following the generation of a parameter configuration by the constraint solver, the `model_gen.py` script directly materializes it into code for a specific framework (PyTorch, TensorFlow, or PaddlePaddle). It uses the framework's native Python API (e.g., `torch.nn.Conv2d`) to construct the operator with the solved parameters, creates tensor objects with the specified shapes, populates them with random data, and executes the operation on the GPU.

Automated Violation Detection. A key technical challenge was reliably automating the detection of memory safety violations. Our main controller script, `controller.py`, uses the `pexpect` library to solve this. It spawns the test case execution script (`model_gen.py`) as a child process, but wraps the entire command with NVIDIA's `compute-sanitizer`. The controller monitors the output for specific error patterns that `compute-sanitizer` emits, such as lines beginning with "`========= Program hit cudaError`", which indicate a GPU memory safety violation. When such a pattern is matched, it terminates the process and archives the logs and the triggering parameters for reproduction.

5 Evaluation

In this section, we evaluate the effectiveness and efficiency of GPU-Fuzz. We aim to answer the following research questions (RQs):

- **RQ1:** How effective is GPU-Fuzz in discovering real-world bugs in major deep learning frameworks?
- **RQ2:** How efficient is the structured generative fuzzing strategy of GPU-Fuzz?

5.1 Experimental Setup

All experiments were conducted on a server with the configuration detailed in Table 2. We established isolated Conda environments for each of the three target frameworks—PyTorch, TensorFlow, and PaddlePaddle—to manage their specific

Table 1: Supported Operator Families and Specific Operators in GPU-Fuzz

Category	Operator Family	Specific Operators	Notes / Examples
Convolution	Convolution	Conv (1d, 2d, 3d) ConvTranspose (1d, 2d, 3d)	Covers 1D, 2D, and 3D standard convolutions. Covers 1D, 2D, and 3D transposed convolutions.
Pooling	Pooling	MaxPool (1d, 2d, 3d) AvgPool (1d, 2d, 3d) FractionalMaxPool (2d, 3d) LPPool AdaptiveAvgPool (1d, 2d, 3d) AdaptiveMaxPool (1d, 2d, 3d)	
Padding	Padding	ConstantPad ReflectionPad ReplicationPad ZeroPad CircularPad	
Element-Wise	Unary Binary	(abs, sin, sqrt, ...) (add, sub, mul, ...)	Includes absolute value, sine, square root, etc. Includes element-wise addition, subtraction, multiplication, etc.
Matrix Ops	MatMul	MatMul / BMM	Standard and batched matrix multiplication.

dependencies. The core hardware, operating system, and NVIDIA driver were consistent across all tests.

Table 2: Experimental Environment Configuration.

Component	Specification
Hardware	
CPU	2 x Intel Xeon Silver 4510 (24 cores / 48 threads total)
GPU	NVIDIA H100 PCIe
Software (Common)	
Operating System	Ubuntu 24.04.2 LTS
NVIDIA Driver	580.82.07
CUDA Runtime	12.8.93
Python	3.11.13
Frameworks (in separate Conda environments)	
PyTorch	PyTorch 2.3.1+cu121 with cuDNN 8.9.2.26
TensorFlow	2.20.0, with cuDNN 9.13.0.50
PaddlePaddle	2.6.1

5.2 Bug Discovery Effectiveness

Over the course of our evaluation, GPU-Fuzz discovered a total of 14 unique, previously unknown bugs across the three frameworks. Table 3 presents a summary of these findings. The bugs span a range of failure modes, from low-level memory corruption to API-level exceptions. We identified 8 distinct memory access violations (e.g., out-of-bounds or misaligned reads/writes). Among these, 6 were silent memory corruptions (poc2, 5, 6, 7, 8, 14) that do not trigger any API-level crash and are only detectable with specialized tools like `compute-sanitizer`. At the time of writing, we have reported all findings by opening issues in the corresponding repositories.

Bug Patterns. Our findings reveal important patterns across three distinct failure modes. (1) The most critical category is *Silent Memory Corruption*, where out-of-bounds or misaligned memory access occurs without causing any API-level error. These are the most insidious bugs as they can lead to silent data corruption and were only detectable with `compute-sanitizer`. (2) The second category is *GPU-Level Exceptions*, where invalid parameters or configurations cause CUDA, cuDNN, or CUBLAS libraries to return an error, which is then typically caught and reported by the framework. (3) The final category is *CPU-Side Asserts*, where issues like integer overflows occur on the CPU during the calculation of kernel parameters, preventing the GPU launch altogether. A common root cause across all frameworks was incorrect grid dimension calculations or flawed boundary checks, with transposed convolutions being particularly error-prone.

Bug Reproduction. To aid in fixing these bugs, GPU-Fuzz logs the parameters, random seed, and input tensors for each test. This enables our scripts to deterministically reproduce the exact failure state, providing developers with a reliable and minimal test case and significantly reducing the time required for analysis.

The results clearly demonstrate that GPU-Fuzz is highly effective at finding critical, real-world bugs. Its ability to generate semantically valid inputs allows it to penetrate the framework's defenses and stress-test the complex, often manually-optimized CUDA code in the backend, which remains a sig-

Table 3: Summary of Bugs Discovered by GPU-Fuzz.

ID	Framework	Operator	Bug Type	Root Cause	Failure Mode
poc1	PyTorch	`conv_transpose2d`	OOB Global Write	Incorrect grid dimension calculation	GPU-Level Exception (CUBLAS)
poc2	PyTorch	`bmm_sparse`	Misaligned Global Write	Incorrect pointer arithmetic in CUSPARSE	Silent Memory Corruption
poc3	PaddlePaddle	`conv2d_transpose`	Precondition Violation	Integer overflow in tensor dimension calc.	CPU-Side Assert
poc4	PaddlePaddle	`conv3d_transpose`	Illegal Instruction	Invalid parameters passed to cuDNN kernel	GPU-Level Exception (cuDNN)
poc5	PyTorch	`adaptive_avg_pool2d`	OOB Global Write	Flawed boundary checks in CUDA kernel	Silent Memory Corruption
poc6	PyTorch	`replication_pad2d`	OOB Global Write	Incorrect grid dimension calculation	Silent Memory Corruption
poc7	PyTorch	`adaptive_max_pool2d`	OOB Global Write	Flawed boundary checks in CUDA kernel	Silent Memory Corruption
poc8/9	TensorFlow	`Conv2D`	OOB Global Read	Incorrect index calculation in kernel	Silent Read / Downstream Crash
poc11	TensorFlow	`Conv2D`	Integer Overflow	Overflow in launch config calculation	CPU-Side Assert
poc12	PaddlePaddle	`conv2d_transpose`	Bad API Parameter	Invalid parameter combination passed to cuDNN	GPU-Level Exception (cuDNN)
poc13	PaddlePaddle	`conv2d_transpose`	Invalid Launch Config	Incorrect grid/block dimension calculation	GPU-Level Exception (CUDA)
poc14	PyTorch	`conv_transpose3d`	OOB Shared Read	Incorrect index calculation for shared memory	Silent Memory Corruption
poc15	PyTorch	`reflection_pad1d`	Invalid Launch Config	Integer overflow in `torch.compile` symint logic	GPU-Level Exception (CUDA)

nificant source of vulnerabilities.

5.3 Efficiency of Constraint-Based Fuzzing

To quantitatively validate our approach, we are conducting a comparative study against NNSmith, a state-of-the-art structure-level fuzzer. Both fuzzers are run under identical conditions with `compute-sanitizer` as the oracle. Our hypothesis is that GPU-Fuzz will detect significantly more GPU memory errors over a fixed period (e.g., 48 GPU-hours), as NNSmith's network-level focus is not designed to explore the specific parameter boundary conditions that trigger these vulnerabilities. This experiment will provide concrete evidence for the complementary nature of parameter-level and structure-level fuzzing.

6 Discussion

While our results demonstrate the effectiveness of GPU-Fuzz, we acknowledge several limitations and areas for future work.

Manual Modeling Effort. The quality of GPU-Fuzz depends on its constraint library. Our current library supports 11 operator families, but this is a subset of the hundreds of operators available. Extending coverage requires manual effort to model constraints (100-150 LoC per family). Future work could explore semi-automating constraint extraction from framework documentation to improve scalability.

Limited Oracle. Our primary oracle, `compute-sanitizer`, excels at finding memory errors but cannot detect silent numerical correctness issues or performance regressions. Differential fuzzing against a trusted CPU implementation is a promising direction for a more comprehensive oracle.

7 Related Work

Our work is positioned at the intersection of deep learning (DL) systems security and software testing, with a specific focus on uncovering memory-related bugs in GPU kernels.

7.1 Fuzzing for Deep Learning Systems

Fuzzing for DL systems primarily involves two approaches: structure-level and API-level testing.

Structure-level fuzzers, including NNSmith [**?**], TVM-Fuzz [**?**], and Lemon [**?**], focus on generating valid neural network models to test the compiler stack. Their strength lies in identifying graph-level optimization bugs. However, their network-centric view is ill-suited for exploring the operator parameter boundaries required to trigger memory access violations in GPU kernels.

API-level fuzzers, such as FreeFuzz [**?**], DeepREL [**?**], and TitanFuzz [**?**], test the API layer by generating valid sequences of function calls. While effective for detecting API-level crashes, they cannot uncover silent memory errors that occur at a lower level and are only visible via specialized tools like `compute-sanitizer`.

In contrast, GPU-FUZZ is a **parameter-level fuzzer**. It complements existing work by focusing on individual operators and systematically exploring their parameter boundary conditions. This targeted approach is uniquely effective at discovering the memory safety vulnerabilities within GPU backends that other methods miss.

7.2 GPU Security

Other research targets the GPU software stack more directly. For example, compiler fuzzers like Cudasmith [**?**] generate random CUDA C programs from scratch to stress-test the CUDA compiler (`nvcc`) itself. In contrast, GPU-FUZZ does not generate new CUDA code; instead, it tests the existing, domain-specific, and highly-optimized CUDA kernels that are handwritten by framework developers to implement core DL operators. Similarly, driver-level fuzzers such as GLeeFuzz [**?**] and Moneta [**?**] test the GPU stack from a different angle. GPU-FUZZ bridges the gap between high-level DL applications and low-level GPU execution by using the frameworks' own operators as the entry point to stress-test the underlying kernels for memory safety.

8 Responsible Disclosure

We have responsibly disclosed all 14 discovered bugs to the respective development teams of PyTorch, TensorFlow, and PaddlePaddle by opening detailed issue reports in their official code repositories. At the time of writing, several of these issues have been acknowledged by the developers, with some leading to active discussions or being incorporated into subsequent patches. We are committed to collaborating with the framework maintainers to help improve the security and robustness of the deep learning ecosystem.

9 Conclusion

In this paper, we presented GPU-FUZZ, a constraint-guided fuzzing system designed to uncover memory-related vulnerabilities in GPU-accelerated deep learning operators. We introduced a parameter-level fuzzing approach to address a critical gap left by existing structure-level fuzzers, which primarily target compiler logic errors. By modeling operator semantics as formal constraints, our system systematically generates valid test cases that probe boundary conditions prone to memory errors.

Our extensive evaluation demonstrates that this constraint-based approach is highly effective. GPU-Fuzz successfully discovered 14 unique, previously unknown bugs in major frameworks like PyTorch, TensorFlow, and PaddlePaddle, the majority of which were critical silent memory errors, undetectable at the Python API level and only discoverable through low-level monitoring.

The success of GPU-FUZZ demonstrates a key takeaway for the security of modern AI systems: comprehensive testing requires a two-pronged approach. Both structure-level fuzzing to secure the compiler stack and parameter-level fuzzing to secure the underlying kernels are necessary and complementary. By making our tool and the discovered bugs publicly available, we hope to provide a valuable resource for developers and researchers working to improve the reliability and security of these foundational software systems.

References