GPU-Fuzz: An Automated System for Fuzzing Deep Learning Frameworks on GPUs

NAME

Abstract

The extensive adoption of GPUs in deep learning (DL) frameworks has made their security and reliability paramount. However, existing fuzzing techniques often fail to penetrate deep into GPU compute kernels because they cannot efficiently generate the semantically valid, complex inputs that DL operators require. This paper introduces GPU-Fuzz, a fuzzing system that overcomes this challenge by leveraging the Z3 SMT solver. GPU-Fuzz automatically generates valid test cases by modeling the intricate parameter and shape relationships of DL operators as formal constraints. Applying this approach to major frameworks like PyTorch, TensorFlow, and PaddlePaddle, we discovered 14 unique, previously unknown bugs. Our system facilitates easy bug reproduction through a comprehensive logging mechanism that captures all necessary parameters and inputs. This work demonstrates that constraint-based fuzzing is a highly effective and practical approach for securing the core components of modern DL systems.

1 Introduction

GPUs have become the cornerstone of modern computing, especially in the realm of artificial intelligence. Deep learning frameworks such as PyTorch, TensorFlow, and PaddlePaddle, which are the core software layer built upon GPUs, directly impact the reliability of countless applications across various domains. The GPU backends of these frameworks are extraordinarily complex, comprising a vast amount of code from low-level libraries like cuDNN and cuBLAS, as well as numerous handwritten CUDA kernels. This complexity makes them a fertile ground for security vulnerabilities and stability issues, where even a minor computation error or memory out-of-bounds access can lead to severe consequences.

However, traditional fuzzing methodologies prove ineffective in this context. The random data streams they generate are almost invariably rejected by the frameworks' stringent input validation checks on tensor shapes and data types. As a result, these fuzzers fail to reach the deeper computational logic within the GPU kernels, leaving a critical attack surface untested.

Our key insight is that to effectively test the GPU backend, the fuzzer must understand and respect the semantic constraints of the operators. For example, the relationship between input and output shapes in a convolutional layer is mathematically defined; generating inputs that adhere to these rules is crucial for bypassing frontend validation and stress-testing the underlying CUDA kernels.

Based on this insight, we have designed and implemented **GPU-Fuzz**, a novel, cross-framework fuzzing system. GPU-Fuzz automates the entire closed-loop process of constraint modeling, solving, test case generation, cross-framework execution, and crash reproduction. It systematically translates the semantic rules of deep learning operators into formal constraints, uses the Z3 SMT solver to generate valid and boundary-aware inputs, and then executes these tests on the GPU backends of major deep learning frameworks.

The main contributions of this paper are as follows:

- We propose and implement GPU-Fuzz, the first system to apply Z3 constraint solving to the problem of cross-framework GPU fuzzing for deep learning applications.
- We demonstrate the effectiveness of our approach by discovering a significant number of previously unknown bugs in widely-used frameworks like PyTorch, TensorFlow, and PaddlePaddle.
- We provide a comprehensive, open-source dataset of real-world bugs discovered by our tool, complete with detailed logs and reproducer scripts, which can serve as a valuable resource for developers and security researchers.
- We argue that our constraint-solving approach is significantly more efficient than traditional random fuzzing by ensuring a high ratio of valid test cases that reach the GPU backend.

2 Preliminary and Motivation

This section provides the necessary technical background and motivation for our research.

2.1 GPU Kernel Error Types

GPU kernels, typically written in CUDA, are susceptible to various errors that can compromise DL frameworks. Our fuzzer targets critical bugs such as illegal memory accesses (where a thread reads/writes out-of-bounds), race conditions from unsynchronized parallel memory access, misaligned memory accesses that trigger hardware exceptions, and integer overflows that can lead to subsequent memory corruption.

2.2 Debugging and Instrumentation Tools

To detect these errors, our work relies on NVIDIA's specialized tools. The NVIDIA `compute-sanitizer` is a powerful tool for detecting memory access errors and race conditions in GPU applications. It functions by instrumenting the GPU code to check memory operations at runtime. When a violation is detected, it terminates the application and generates a detailed report. In GPU-Fuzz, `compute-sanitizer` serves as our primary test oracle for detecting memory corruption.

2.3 Motivation

Our research is driven by two core observations: the inefficiency of existing fuzzers and the irreproducibility of GPU bugs.

The Inefficiency of Existing Fuzzers. A fundamental "semantic gap" challenges the fuzzing of DL frameworks. APIs expect highly structured inputs with strict constraints on data types and tensor shapes.

The Irreproducibility of GPU Bugs. Even when a GPU crash is triggered, reproducing it manually is often a time-consuming and arduous process. A typical bug report might only contain a stack trace, but the crash itself is often highly dependent on the specific inputs, tensor shapes, and operator parameters that triggered the illegal state. Without a systematic way to record the exact inputs and parameters, debugging and fixing these issues becomes a significant engineering burden.

These observations motivate the design of GPU-Fuzz. We need a system that can bridge the semantic gap by generating valid, boundary-aware inputs to effectively test GPU kernels. Furthermore, we need a fully automated system that not only finds crashes but also provides the necessary fuzzing information through logs to assist in reproducing the discovered bugs, drastically improving the efficiency of the debugging workflow.

3 System Design

This section details the architecture of GPU-Fuzz. As illustrated in Figure **??**, 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.

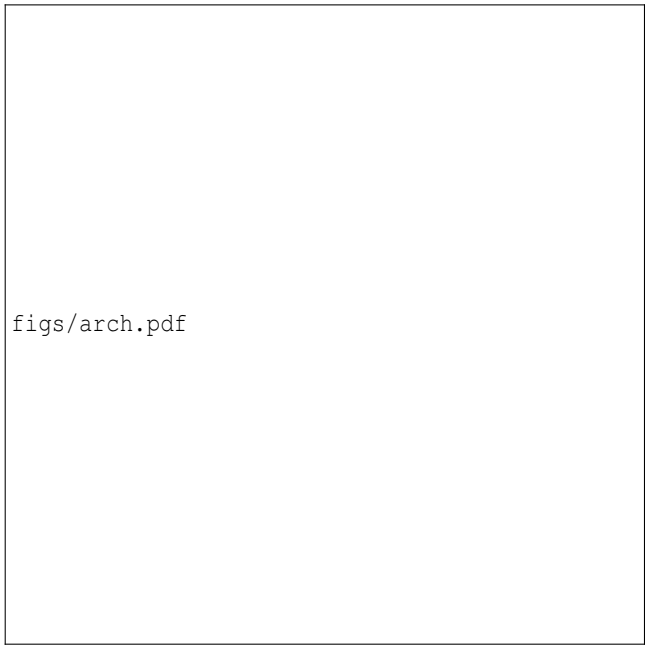

Figure 1: The architecture of the GPU-Fuzz system.

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 are expressed using the Z3 SMT solver's Python API, creating a framework-agnostic representation of the operator's contract. This approach ensures that any test case generated by the solver is, by construction, guaranteed to pass the framework's initial validation checks.

3.2 Constraint Solving and 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 the Z3 solver. The solver's task is to find a concrete assignment of values that satisfies all constraints. The result is a set of valid parameters (e.g., `input_shape`, `kernel_size`, `stride`) and the corresponding output tensor shape. This entire configuration is saved as a JSON file, representing an abstract, executable test case.

3.3 Cross-Framework Execution and Monitoring

Once a set of valid parameters is generated, the test case 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 bug is detected, `compute-sanitizer` terminates the process and generates a detailed report, which serves as our primary bug oracle.

3.4 Log-based Reproduction

A critical component of GPU-Fuzz is its comprehensive logging system, designed to ensure bug reproducibility. For every test case, the fuzzer records a detailed JSON log containing:

- The target operator and framework.
- The exact set of parameters used (e.g., kernel size, stride, padding).
- The shapes of all input tensors.
- The random seed used for data generation and any other stochastic choices.

In addition to the JSON log, the complete input tensors are saved to separate files. When a crash occurs, this combination of logs provides a complete, self-contained record of the conditions that triggered the bug. A developer can then use a simple script to load these artifacts and deterministically reproduce the failure, significantly streamlining the debugging process.

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. Table **??** lists the supported operator families.

Building a model for a new operator family, such as `AbsConv`, typically requires 100-150 lines of code to define its parameters as Z3 symbolic variables and encode the mathematical relationships between them (e.g., the output size formula) as Z3 constraints. The `materialize` method in each operator class then iterates through valid solutions found by the Z3 solver.

Framework Instantiation. The parameters generated by the Z3 solver are saved to a JSON file. The `model_gen.py` script reads this file and materializes the abstract test case 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.

Crash Detection with Compute Sanitizer. A key technical challenge was reliably automating crash detection. 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 upon detecting a GPU kernel error. When such a pattern is matched, it terminates the process and archives the logs 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?
- **RQ3:** How does GPU-Fuzz's log-based approach facilitate bug reproduction and analysis?

5.1 Experimental Setup

All experiments were conducted on a server with the configuration detailed in Table **??**. We established isolated Conda environments for each of the three target frameworks—PyTorch, TensorFlow, and PaddlePaddle—to manage their specific dependencies. The core hardware, operating system, and NVIDIA driver were consistent across all tests.

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.

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 RQ1: 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 **??** presents a summary of these findings. The bugs span a wide range of types, from memory corruption errors deep within CUDA kernels to integer overflows and incorrect API parameter handling at the framework level. Notably, 12 of these bugs were silent errors that would not have caused a crash at the Python level and were only detectable thanks to `compute-sanitizer`. At the time of writing, we have reported all findings by opening issues in the corresponding repositories.

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 significant source of vulnerabilities.

5.3 RQ2: Efficiency of Constraint-Based Fuzzing

GPU-Fuzz's constraint-based approach is designed to be more efficient than naive random fuzzing. By modeling the valid parameter space of an operator using Z3, we ensure that every generated test case is semantically correct and passes the framework's initial API-level validation checks. This allows the fuzzer to focus its resources on executing code in the GPU backend, which is the primary target of our investigation, rather than wasting cycles on invalid inputs that are immediately rejected.

While a quantitative comparison against a baseline random fuzzer was not conducted in this study, the logical advantage is clear. The parameter space for deep learning operators is vast and sparse; for instance, the relationships between input shape, kernel size, stride, and padding in a convolution are strict. A naive fuzzer would have a vanishingly small probability of generating a valid combination. In contrast, GPU-Fuzz achieves a near-100% rate of valid test cases by construction, making it a far more practical approach for discovering deep bugs in a reasonable timeframe. A full quantitative evaluation of this efficiency gain remains an important direction for future work.

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

ID	Framework	Operator	Bug Type	Root Cause	Note
poc1	PyTorch	`conv_transpose2d`	OOB Global Write	Incorrect grid dimension calculation	CUBLAS fail
poc2	PyTorch	`bmm_sparse`	Misaligned Global Write	Incorrect pointer arithmetic in CUSPARSE	Silent error
poc3	PaddlePaddle	`conv2d_transpose`	Precondition Violation	Integer overflow in tensor dimension calc.	CPU-side error
poc4	PaddlePaddle	`conv3d_transpose`	Illegal Instruction	Invalid parameters passed to cuDNN kernel	CUDNN fail
poc5	PyTorch	`adaptive_avg_pool2d`	OOB Global Write	Flawed boundary checks in CUDA kernel	Silent error
poc6	PyTorch	`replication_pad2d`	OOB Global Write	Incorrect grid dimension calculation	Silent error
poc7	PyTorch	`adaptive_max_pool2d`	OOB Global Write	Flawed boundary checks in CUDA kernel	Silent error
poc8/9	TensorFlow	`Conv2D`	OOB Global Read	Incorrect index calculation in kernel	Silent, downstream fails
poc11	TensorFlow	`Conv2D`	Integer Overflow	Overflow in launch config calculation	CPU assertion fail
poc12	PaddlePaddle	`conv2d_transpose`	Bad API Parameter	Invalid parameter combination passed to cuDNN	CUDNN_STATUS_BAD_PARAM
poc13	PaddlePaddle	`conv2d_transpose`	Invalid Launch Config	Incorrect grid/block dimension calculation	`cudaErrorInvalidConfiguration`
poc14	PyTorch	`conv_transpose3d`	OOB Shared Read	Incorrect index calculation for shared memory	Silent error
poc15	PyTorch	`reflection_pad1d`	Invalid Launch Config	Integer overflow in `torch.compile` symint logic	`cudaErrorInvalidConfiguration`

5.4 RQ3: Log-based Bug Reproduction

A key challenge in fixing GPU bugs is the difficulty of reproduction. To address this, GPU-Fuzz implements a comprehensive logging mechanism that facilitates a semi-automated reproduction workflow. For every test case executed, the system saves a detailed log containing the exact operator, all input parameters, the random seed used for any internal stochastic decisions, and the full input tensors.

When a crash is detected by an external tool like `compute-sanitizer`, these logs provide all the necessary information to deterministically reproduce the failure. We have developed reproducer scripts (located in the `poc/` directory) that read these log files. By using the logged random seed and parameters, the script can faithfully reconstruct the exact state that triggered the bug, providing a reliable and minimal test case for developers. This process significantly reduces the time and effort required for bug analysis, bridging the gap from initial crash detection to root-cause analysis.

6 Discussion and Limitations

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

Manual Modeling Effort. The effectiveness of GPU-Fuzz is contingent on the quality and coverage of its constraint library. As detailed in Section **??**, modeling a new operator family requires manual effort to encode its constraints in Z3. While this one-time cost is amortized over long fuzzing campaigns, it presents a scalability challenge for covering the hundreds of operators available in modern deep learning frameworks. Future work could explore techniques to semi-automate the extraction of constraints from framework documentation or operator implementations.

Lack of Quantitative Efficiency Comparison. In our evaluation of RQ2, we argued for the efficiency of our approach from a design standpoint but did not provide a quantitative comparison against a baseline random fuzzer. Such an experiment, measuring metrics like valid test cases per hour or time-to-first-bug, would provide a more rigorous validation of the performance benefits of constraint-based fuzzing. We consider this a key priority for future work.

Limited Oracle. Our primary test oracle is `compute-sanitizer`, which is excellent for finding memory corruption bugs. However, it cannot detect other classes of bugs, such as silent numerical correctness issues (where the kernel produces the wrong result without crashing) or performance regressions. A more comprehensive approach would involve differential fuzzing against a CPU implementation, which is a promising direction for future work.

7 Related Work

The security of deep learning (DL) frameworks is a critical research area. Our work is positioned at the intersection of fuzzing for DL frameworks and GPU systems.

Prior work in DL fuzzing has focused on bridging the semantic gap to generate valid inputs. While API-level fuzzers like **FreeFuzz** [**?**] and **DeepREL** [**?**] learn API parameters from code, and LLM-driven fuzzers like **TitanFuzz** [**?**] and **FlashFuzz** [**?**] generate plausible high-level code, they often miss the boundary conditions that our constraint-solving approach is designed to systematically explore. GPU-Fuzz distinguishes itself by adopting a formal methods approach. It translates the mathematical rules of DL operators into formal constraints and uses an SMT solver (Z3) to generate inputs that are provably valid. This allows for exceptional effectiveness in stress-testing the low-level GPU compute kernels, which is our primary target.

Other research targets the GPU software stack directly. Compiler fuzzers like **CUDAsmith** [**?**] and **GraphicsFuzz** [**?**] are vital for ensuring toolchain integrity but do not test the hand-written CUDA kernels within DL frameworks. Similarly, driver-level fuzzers such as **GLeeFuzz** [**?**] and **Moneta** [**?**] test the GPU stack from a different angle. In contrast, GPU-Fuzz specifically targets the attack surface exposed by high-level DL operators, using the frameworks as a vehicle to

reach the deep computational logic within the GPU kernels.

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 novel fuzzing framework that leverages the Z3 SMT solver to discover bugs in deep learning GPU kernels. Our work tackles the critical challenge of the "semantic gap" by generating constraint-satisfying, valid inputs that can effectively exercise the deep backend logic of DL frameworks.

Our extensive evaluation demonstrates that this constraint-based approach is both effective and efficient. GPU-Fuzz successfully discovered 14 unique, previously unknown bugs in major frameworks like PyTorch, TensorFlow, and PaddlePaddle. Furthermore, we showed that our log-based reproduction methodology drastically reduces the time required for bug analysis and debugging.

The bugs found by GPU-Fuzz underscore the continued need for specialized, constraint-aware fuzzing tools in the domain of deep learning. 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 modern AI systems.

References