GPU-Fuzz: Finding Memory Errors in Deep Learning Frameworks

ABSTRACT

GPU memory errors are a critical threat to deep learning (DL) frameworks, leading to crashes or even security issues. We introduce GPU-Fuzz, a fuzzer addressing this issue by modeling operator parameters as formal constraints. GPU-Fuzz utilizes a constraint solver to generate test cases that systematically probe error-prone boundary conditions in GPU kernels. Applied to PyTorch, TensorFlow, and PaddlePaddle, we uncovered 13 unknown bugs, which demonstrate the effectiveness of GPU-Fuzz in finding memory errors.

ACM Reference Format:

. 2025. GPU-Fuzz: Finding Memory Errors in Deep Learning Frameworks. In . ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

GPUs are now an indispensable component of deep learning (DL) frameworks like PyTorch [21] and TensorFlow [1]. However, the correctness of GPU computations are often threatened by memory corruptions, an insidious class of bugs stemming from the low-level CUDA kernels [18]. 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 [9], posing a significant threat to the reliability and security of AI applications.

However, locating these memory-related bugs remains a profound challenge. Existing fuzzers for DL systems are primarily designed to find arithmetic miscomputions in the DL compilers by generating diverse neural networks with different structures [15]. This approach, while effective for compiler testing, is ill-suited for uncovering memory errors due to the lack of exploration of the operator parameter space.

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 patterns and generate inputs that systematically explore the parameter space.

To this end, we designed and implemented GPU-Fuzz, a novel fuzzer specifically engineered to locate these memory-related bugs. Unlike structure-oriented fuzzers such as NNSmith [15], 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 diversified inputs that probe memory-related boundary conditions. This constraint-guided approach [3] enables GPU-Fuzz to effectively stress-test the low-level CUDA kernels for memory safety.

The main contributions of this paper are as follows:

- We propose a new fuzzing approach that targets GPU memory errors by systematically exploring the operator parameter space, a dimension orthogonal to existing DL fuzzers [15].
- We design and implement GPU-Fuzz, a system that leverages 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 uncovering 13 previously unknown bugs, in major DL frameworks like PyTorch.

2 BACKGROUND AND MOTIVATION

2.1 GPU Architecture

Modern GPUs are massively parallel processors designed for high-throughput computation. Their architecture is built around a collection of Streaming Multiprocessors (SMs), each capable of executing hundreds of threads concurrently. The CUDA programming model abstracts this hardware [18], organizing threads into a hierarchy of grids, blocks, and warps. This parallelism is supported by a complex memory hierarchy [12], comprising high-bandwidth global memory, low-latency shared memory private to each SM, and per-thread local memory and registers. Writing efficient GPU code requires developers to manually manage data placement and movement across this hierarchy [10]. The complexity of this task, especially the intricate pointer arithmetic required for optimal memory access patterns, makes GPU kernels highly susceptible to memory errors such as out-of-bounds access and race conditions [13].

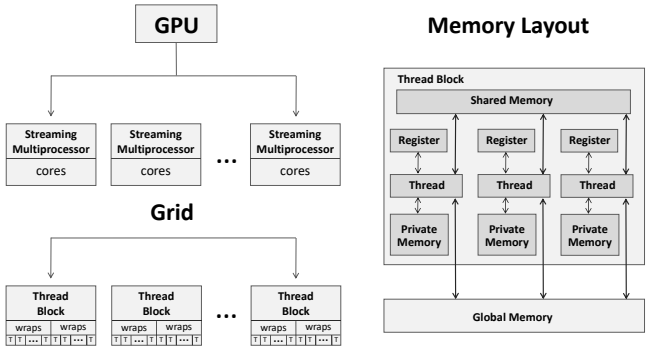

Fig. 1: GPU architecture and memory layout.

2.2 Deep Learning Operators

Deep learning (DL) frameworks like PyTorch [21] and TensorFlow [1] are built around a rich library of fundamental computational units called operators. These operators, such as convolution, pooling, matrix multiplication, and activation functions [5], serve as the elemental building blocks for constructing neural networks.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

While users interact with them through simple, high-level Python APIs, the underlying reality is far more complex. Each operator is backed by one or more highly-optimized, low-level programs known as kernels [5], which are executed on the GPU.

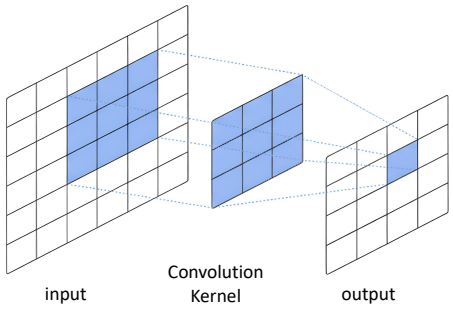

Fig. 2: An illustration of a convolution operator.

The complexity arises from the vast parameter space of each operator. A single convolution operator [8], for instance, is governed by a multitude of parameters beyond the input tensor itself: kernel size, stride, padding, dilation, and channel groups. These parameters are not independent; they are bound by a complex set of semantic and mathematical constraints that dictate valid combinations and determine the output tensor's properties. To achieve state-of-the-art performance, framework developers implement these kernels manually in CUDA C++, employing sophisticated techniques like shared memory tiling and intricate pointer arithmetic to maximize data throughput [10]. This manual, performance-driven optimization often bypasses safer, high-level abstractions, making the kernel code a fertile ground for subtle memory errors [13] that are triggered only by specific, often obscure, parameter configurations.

2.3 Motivation

GPU memory bugs represent a severe and often silent threat to the reliability and security of AI systems [20]. These bugs can cause catastrophic failures in mission-critical applications like medical imaging [23] and autonomous driving [14], or be exploited for security attacks [22].

State-of-the-art DL fuzzers focus on the compiler stack, generating valid neural networks to find bugs [15, 16]. This network-level approach is ill-suited for finding low-level memory errors in GPU kernels. Such bugs are not typically triggered by network architecture, but by specific, often boundary-value, combinations of an operator's parameters (e.g., tensor shapes, strides). This leaves a fundamental blind spot: existing fuzzers like NNSmith [15] do not systematically explore the intricate parameter space of individual operators where these memory bugs reside.

This observation reveals the need for a paradigm shift toward operator-level fuzzing. We introduce GPU-Fuzz, a system designed to explore the operator parameter space to uncover memory bugs in low-level CUDA kernels.

3 SYSTEM DESIGN

This section details the architecture of GPU-Fuzz. As illustrated in Figure 3, the system consists of three main phase, operator modeling, constraint-based testcase generation and cross-framework execution.

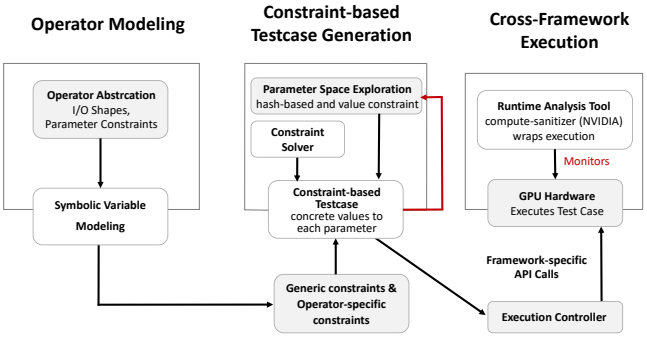

Fig. 3: The architecture of the GPU-Fuzz system.

3.1 Operator modeling.

GPU-Fuzz models GPU operators through an abstraction layer that captures their parameter spaces and shape relationships. Each operator family (e.g., convolution, pooling, activation) is represented by a unified model that defines the interface for input/output shapes and parameter constraints. For operators with complex shape transformations, such as convolutions and pooling operations, we introduce symbolic variables using the Z3 SMT solver [7] to represent operator parameters. The abstraction allows us to model operators uniformly across different frameworks while preserving framework-specific semantics.

3.2 Constraint-based Testcase Generation

Constraint solving.GPU-Fuzz encodes operator semantics as constraints over symbolic variables through two layers: generic constraints and operator-specific constraints.

Generic constraints ensure basic validity: operator parameters (e.g., tensor dimensions) must be reasonable positive integers, and total I/O tensor memory must not exceed GPU limits.

Operator-specific constraints capture the mathematical relationships between input and output shapes. Figure 4 illustrates that for a convolution operator, the output size along each spatial dimension follows the formula [8]:

$$H_{\text{out}} = \frac{H_{\text{in}} + 2P - D(K-1) - 1}{S} + 1$$

where H_{in} and H_{out} denote the input and output sizes, P is padding, D is dilation, K is kernel size, and S is stride. Additional constraints ensure semantic correctness, such as requiring the input size to be larger than the kernel size.

GPU-Fuzz uses Z3's SMT solver [7] to find satisfying assignments for all symbolic variables. The solver returns a model that assigns concrete values to each parameter, which are then used to instantiate the operator for testcase generation.

Parameter space exploration. To systematically explore the parameter space, GPU-Fuzz employs an iterative constraint-guided search strategy. After obtaining a solution from the solver, the system randomly selects one symbolic variable from the current model and adds constraints that exclude its current value. Specifically, the system adds both a hash-based constraint and a direct value constraint: the hash constraint helps the solver efficiently prune value ranges with better dispersion, while the direct constraint

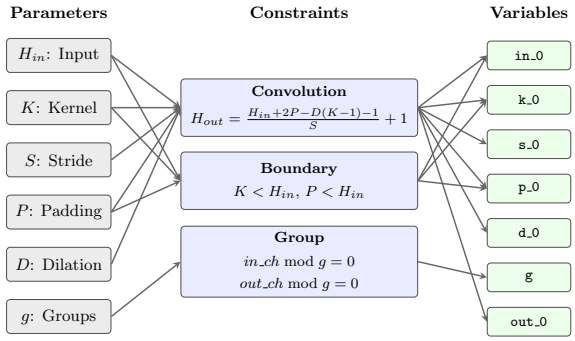

Fig. 4: Constraint modeling for convolution operators.

ensures the new solution differs from the current one. This dual-constraint approach improves solver efficiency by providing better value distribution guidance, while the incremental constraint addition forces the solver to find different solutions in subsequent iterations, effectively guiding the search toward unexplored regions of the parameter space. Figure 5 illustrates this process.

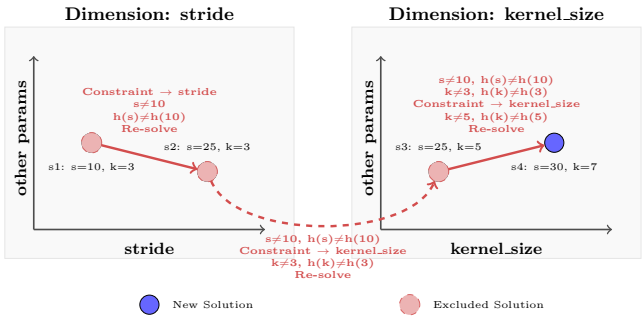

Fig. 5: An example of the iterative parameter space exploration. At each step, a parameter is randomly selected and a new constraint that excludes its current value is incrementally added to guide the search for the next solution.

3.3 Cross-framework Execution

GPU-Fuzz detects framework-specific GPU bugs by executing generated test cases across multiple deep learning frameworks (PyTorch [21], PaddlePaddle [17], and TensorFlow [1]). Its execution pipeline involves materializing abstract operator representations into framework-specific API calls, handling nuances like parameter naming and data formats. An execution controller orchestrates this by iteratively generating configurations, materializing, and executing them on GPUs. NVIDIA Compute Sanitizer [19] wraps each execution to monitor for GPU memory errors and kernel failures (e.g., out-of-bounds). This cross-framework approach facilitates differential testing, identifying inconsistencies and framework-specific bugs by executing semantically equivalent test cases across different frameworks.

4 IMPLEMENTATION

This section details the implementation of the system design described in Section 3. Our system is developed in Python and comprises approximately 2,000 lines of code.

Tab. 1: Supported Operator Families in GPU-Fuzz

Operator Family	Specific Operators
Convolution	Conv (1d, 2d, 3d), ConvTranspose (1d, 2d, 3d)
Pooling	MaxPool (1d, 2d, 3d), AvgPool (1d, 2d, 3d), FractionalMaxPool (2d, 3d), LPPool, AdaptiveAvgPool (1d, 2d, 3d), AdaptiveMaxPool (1d, 2d, 3d)
Padding	ConstantPad, ReflectionPad, ReplicationPad, ZeroPad, CircularPad
Element-Wise Unary	abs, sin, sqrt, ...
Element-Wise Binary	add, sub, mul, ...
Matrix Ops	MatMul, BMM

We implemented a library where each operator is represented as a distinct class to realize the operator modeling and constraint-based testcase generation concepts described in Section 3. This modeling approach was applied to 11 operator families (Table 1), chosen for their prevalence in deep learning models and their complex, error-prone memory access patterns.

An execution controller translates the generated parameter configurations into executable test cases for PyTorch [21], TensorFlow [1], and PaddlePaddle [17]. The execution is wrapped with NVIDIA's compute-sanitizer [19]. When it detects an error, the controller archives the logs and the triggering operator configuration to ensure reproducibility.

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 uncovering real-world bugs in major deep learning frameworks?
- **RQ2:** How does GPU-Fuzz compare with state-of-the-art structure-level fuzzers in terms of test case generation and bug uncovery, particularly for GPU memory errors?

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

Tab. 2: Experimental Environment Configuration.

Component	Specification
Hardware	
CPU	2 x Intel Xeon Silver 4510
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 uncovered a total of 13 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 (Bug_2, Bug_5, Bug_6, Bug_7, Bug_8, Bug_{12}) that do not trigger any API-level crash and are only detectable with specialized tools like compute-sanitizer [19]. 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:

- **Silent Memory Corruption:** The most critical category, 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 are only detectable with low-level memory debuggers.
- **GPU-Level Exceptions:** The second category, 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.
- **CPU-Side Asserts:** The final category, 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. Figure 6 visualizes these patterns by error type.

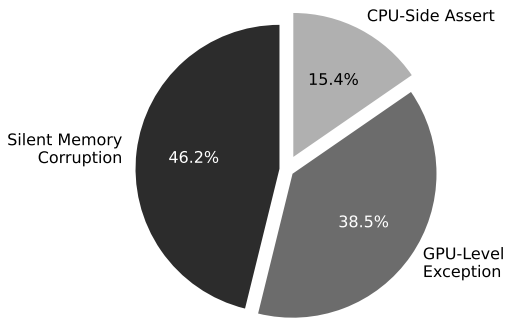

Fig. 6: Bug distribution statistics by error type.

5.3 Comparative Study

To quantitatively validate our approach, we conducted a comparative study against NNSmith [15], a state-of-the-art structure-level fuzzer. We conducted five independent 4-hour fuzzing runs for each tool on the same hardware targeting PyTorch, with both tools running in identical Conda environments.

Table 4 summarizes the results. NNSmith generated on average 19,063 ± 360 test cases and uncovered 296 ± 19 bugs. Most of its findings are numerical mismatches rather than memory-safety issues. In contrast, GPU-Fuzz generated on average 51,860 ± 1,559 test cases and uncovered 106 ± 8 real bugs excluding out-of-memory errors, including 26 ± 5 critical memory errors and 81 ± 7 configuration errors. These memory errors represent severe security vulnerabilities that could result in data corruption, information leakage, or system crashes.

Key Findings. Our analysis reveals two critical insights. First, GPU-Fuzz generated nearly three times more test cases than NNSmith, demonstrating the efficiency of constraint-guided parameter fuzzing in systematically exploring the operator parameter space. Second, and more importantly, GPU-Fuzz uncovered 26 ± 5 critical memory errors that pose serious security risks, while NNSmith's findings were primarily numerical precision issues that typically do not threaten memory safety. This demonstrates that GPU-Fuzz addresses a critical blind spot in GPU memory security testing that structure-level fuzzers frequently miss. The two approaches are complementary: NNSmith excels at uncovering compiler-related bugs and numerical inconsistencies at the network structure level, while GPU-Fuzz fills the gap in GPU memory security testing at the operator parameter level.

5.4 Case Study

To illustrate the practical utility of GPU-Fuzz, we present a minimal proof-of-concept (PoC) for a memory corruption bug uncovered in PyTorch's `ConvTranspose2d` operator. The PoC is shown in Figure 7.

```
import torch
D, C = 40000, 10
m1 = torch.randn(C, D, 2).cuda()
model = torch.nn.ConvTranspose2d(
    C, 2, kernel_size=(1, 1),
    stride=(200, 200)).cuda()
model(m1)
```

Fig. 7: A minimal PoC that triggers a memory corruption bug in PyTorch's `ConvTranspose2d`.

The key to triggering this bug lies in the parameter combination automatically generated by our fuzzer, particularly the extremely large stride value of `(200, 200)`. While this value is semantically valid according to PyTorch's API, it represents a corner case that is unlikely to be covered by manual unit tests. When this code is executed, the low-level oracle in our system detects a cascade of out-of-bounds global write violations, which are critical memory-safety errors. Figure 8 shows an excerpt from the terminal output, highlighting one such violation. The detailed error log reveals that these out-of-bounds memory accesses occur within the low-level `col2im_kernel` CUDA kernel, which is fundamental to the transposed convolution operation. This type of silent memory corruption is a severe bug that can lead to incorrect results or unpredictable behavior without causing an explicit API-level crash. This case demonstrates the ability of GPU-Fuzz to systematically uncover severe, hidden bugs in mature deep learning frameworks by exploring non-trivial parameter spaces.

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,

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

ID	Framework	Operator	Bug Type	Root Cause	Failure Mode
$\mathbf{Bug_1}$	PyTorch	`conv_transpose2d`	OOB Global Write	Incorrect grid dimension calculation	GPU-Level Exception (CUBLAS)
$\mathbf{Bug_2}$	PyTorch	`bmm_sparse`	Misaligned Global Write	Incorrect pointer arithmetic in CUSPARSE	Silent Memory Corruption
$\mathbf{Bug_3}$	PaddlePaddle	`conv2d_transpose`	Precondition Violation	Integer overflow in tensor dimension calculation	CPU-Side Assert
$\mathbf{Bug_4}$	PaddlePaddle	`conv3d_transpose`	Illegal Instruction	Invalid parameters passed to cuDNN kernel	GPU-Level Exception (cuDNN)
$\mathbf{Bug_5}$	PyTorch	`adaptive_avg_pool2d`	OOB Global Write	Flawed boundary checks in CUDA kernel	Silent Memory Corruption
$\mathbf{Bug_6}$	PyTorch	`replication_pad2d`	OOB Global Write	Incorrect grid dimension calculation	Silent Memory Corruption
$\mathbf{Bug_7}$	PyTorch	`adaptive_max_pool2d`	OOB Global Write	Flawed boundary checks in CUDA kernel	Silent Memory Corruption
$\mathbf{Bug_8}$	TensorFlow	`Conv2D`	OOB Global Read	Incorrect index calculation in kernel	Silent Read / Downstream Crash
$\mathbf{Bug_9}$	TensorFlow	`Conv2D`	Integer Overflow	Overflow in launch config calculation	CPU-Side Assert
$\mathbf{Bug_{10}}$	PaddlePaddle	`conv2d_transpose`	Bad API Parameter	Invalid parameter combination passed to cuDNN	GPU-Level Exception (cuDNN)
$\mathbf{Bug_{11}}$	PaddlePaddle	`conv2d_transpose`	Invalid Launch Config	Incorrect grid/block dimension calculation	GPU-Level Exception (CUDA)
$\mathbf{Bug_{12}}$	PyTorch	`conv_transpose3d`	OOB Shared Read	Incorrect index calculation for shared memory	Silent Memory Corruption
$\mathbf{Bug_{13}}$	PyTorch	`reflection_pad1d`	Invalid Launch Config	Integer overflow in `torch.compile` symint logic	GPU-Level Exception (CUDA)

Tab. 4: Comparative Results

Metric	NNSmith	GPU-Fuzz
Test Cases Generated	19,063 ± 360	51,860 ± 1,559
Total Bugs*	296 ± 19	106 ± 8
Bug Breakdown by Type		
Memory Errors	0	26 ± 5
Configuration Errors	0	81 ± 7
Inconsistencies	293 ± 19	0
Exceptions	3 ± 1	0
Runtime	4 GPU-hours each	

* GPU-Fuzz total excludes out-of-memory errors.
** NNSmith and GPU-Fuzz results are mean ± std over 5 independent runs.

```
========= Invalid __global__ write of size 4 bytes
=========    at void at::native::col2im_kernel<...>(...)
=========    by thread (0,0,0) in block (4194446,0,0)
=========    Address 0x7ff778047000 is out of bounds
[...]
=========        Host Frame: <module> in poc1.py:7
```

Fig. 8: The error message for the PoC in Figure 7

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 can be categorized into structure-level, library-level, and API-level testing approaches [24].

Structure-level fuzzers, including NNSmith [15], TVMFuzz [24], and HirGen [16], focus on generating valid neural network models to test the compiler stack (e.g., TVM [4]). These tools are effective at identifying compiler-related bugs, such as graph-level optimization issues and IR transformation errors. However, their network-centric view is not designed to systematically explore the operator parameter boundaries required to trigger memory access violations in GPU kernels.

Library-level fuzzers, such as LEMON [25], test DL library implementations by generating model variants to detect inconsistencies across different libraries. While effective for finding library-specific bugs, they operate at a higher abstraction level and are not designed to systematically probe the low-level memory access patterns within GPU kernels.

API-level fuzzers, such as Orion [11], test the API layer by generating test inputs guided by historical bug patterns. While effective for detecting API-level crashes and certain classes of bugs, they typically 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 effective at uncovering memory safety vulnerabilities within GPU backends that may be missed by other methods.

7.2 GPU Security

Research on GPU security can be categorized into several approaches.

Static verification tools, such as GPUVerify [2], employ formal methods to verify GPU kernel correctness by detecting data races and barrier divergence. While effective for general-purpose GPU kernels, these tools require source code access and are not tailored to the domain-specific operators used in DL frameworks.

Runtime detection tools, such as iGUARD [13], detect data races during execution through runtime monitoring using binary instrumentation (NVBit). While iGUARD can detect certain memory-related concurrency issues, it requires runtime instrumentation and primarily focuses on data race detection rather than general memory access violations such as out-of-bounds access.

Compiler fuzzers, such as DeepSmith [6], generate random CUDA C programs from scratch to stress-test the CUDA compiler (`nvcc`) [10] 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.

Multi-tenant security research, such as Guardian [22], focuses on providing memory isolation in multi-tenant GPU environments. While addressing an important security concern, these approaches do not directly detect memory errors within individual kernels.

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. Unlike static verification tools, GPU-Fuzz requires no source code access and works directly with compiled frameworks. Unlike runtime detection tools, it uses external oracles (compute-sanitizer) without modifying the runtime environment. Unlike compiler fuzzers, it targets domain-specific DL operators rather than generic CUDA programs.

8 CONCLUSION

We introduced GPU-Fuzz, a constraint-guided fuzzer that aims to address the challenge of finding memory errors in deep learning operators. By shifting the focus from network structure to the operator parameter level, GPU-Fuzz explores boundary conditions that may trigger low-level vulnerabilities. Our approach was able to uncover 13 previously unknown bugs in widely used frameworks such as PyTorch, TensorFlow, and PaddlePaddle, most of which were silent memory errors. This work suggests that securing modern AI systems may benefit from a complementary strategy combining both structure-level and parameter-level fuzzing. We hope that by making our tool and findings publicly available, we can contribute to improving the reliability and security of these foundational technologies.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. {TensorFlow}: a system for {Large-Scale} machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283, 2016.

[2] Adam Betts, Nathan Chong, Alastair Donaldson, Shaz Qadeer, and Paul Thomson. Gpuverify: a verifier for gpu kernels. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pages 113–132, 2012.

[3] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[4] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, 2018.

[5] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[6] Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. Compiler fuzzing through deep learning. In *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*, pages 95–105, 2018.

[7] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[8] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning. *arXiv preprint arXiv:1603.07285*, 2016.

[9] David Fiala, Frank Mueller, Christian Engelmann, Rolf Riesen, Kurt Ferreira, and Ron Brightwell. Detection and correction of silent data corruption for large-scale high-performance computing. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2012.

[10] Design Guide. Cuda c++ programming guide. *NVIDIA, July*, 2020.

[11] Nima Shiri Harzevili, Mohammad Mahdi Mohajer, Moshi Wei, Hung Viet Pham, and Song Wang. History-driven fuzzing for deep learning libraries. *ACM Trans. Softw. Eng. Methodol.*, 34(1):19–1, 2025.

[12] Sunpyo Hong and Hyesoon Kim. An integrated gpu power and performance model. In *Proceedings of the 37th annual international symposium on Computer architecture*, pages 280–289, 2010.

[13] Aditya K Kamath and Arkaprava Basu. Iguard: In-gpu advanced race detection. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 49–65, 2021.

[14] Alex H Lang, Sourabh Vora, Holger Caesar, Lubing Zhou, Jiong Yang, and Oscar Beijbom. Pointpillars: Fast encoders for object detection from point clouds. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 12697–12705, 2019.

[15] Jiawei Liu, Jinkun Lin, Fabian Ruffy, Cheng Tan, Jinyang Li, Aurojit Panda, and Lingming Zhang. Nnsmith: Generating diverse and valid test cases for deep learning compilers. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 530–543, 2023.

[16] Haoyang Ma, Qingchao Shen, Yongqiang Tian, Junjie Chen, and Shing-Chi Cheung. Fuzzing deep learning compilers with hirgen. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 248–260, 2023.

[17] Yanjun Ma, Dianhai Yu, Tian Wu, and Haifeng Wang. Paddlepaddle: An open-source deep learning platform from industrial practice. *Frontiers of Data and Domputing*, 1(1):105–115, 2019.

[18] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda: Is cuda the parallel programming model that application developers have been waiting for? *Queue*, 6(2):40–53, 2008.

[19] NVIDIA Corporation. *NVIDIA Compute Sanitizer User Guide*. NVIDIA Corporation, version 2023.3 edition, 2023. https://docs.nvidia.com/cuda/compute-sanitizer/.

[20] George Papadimitriou and Dimitris Gizopoulos. Silent data corruptions: Microarchitectural perspectives. *IEEE Transactions on Computers*, 72(11):3072–3085, 2023.

[21] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

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

[23] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.

[24] Qingchao Shen, Haoyang Ma, Junjie Chen, Yongqiang Tian, Shing-Chi Cheung, and Xiang Chen. A comprehensive study of deep learning compiler bugs. In *Proceedings of the 29th ACM Joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pages 968–980, 2021.

[25] Zan Wang, Ming Yan, Junjie Chen, Shuang Liu, and Dongdi Zhang. Deep learning library testing via effective model generation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 788–799, 2020.