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 locating these issues efficiently 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, demonstrating 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, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

GPUs are now an indispensable component of deep learning (DL) frameworks like PyTorch [22] and TensorFlow [1]. However, the correctness of GPU computations is 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 [7], 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 miscomputations 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 existing DL 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 [8] 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, TensorFlow, and PaddlePaddle.

Conference'17, July 2017, Washington, DC, USA
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 BACKGROUND AND MOTIVATION

2.1 GPU Architecture

Modern GPUs are massively parallel devices built on Streaming Multiprocessors (SMs), each executing hundreds of threads concurrently [18]. This parallelism is supported by a complex memory hierarchy [13]. However, harnessing this hardware for performance requires developers to manually manage data placement and movement [9]. The complexity of this manual task makes GPU kernels highly susceptible to memory errors such as out-of-bounds access and race conditions [12].

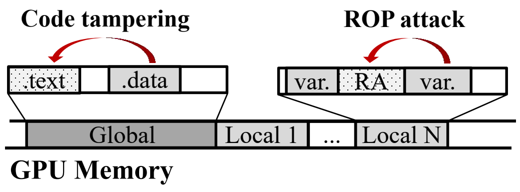


Fig. 1: GPU memory layout and potential attacks.

As illustrated in Fig. 1, these memory errors can be exploited to compromise the kernel's integrity and control flow. The memory spaces most relevant to these attacks are:

Global Memory. The global memory is a large memory space shared by all threads. It is commonly used to store the kernel's executable instructions and global variables. Since previous works have shown that GPUs do not support W⊕X permission [10], an out-of-bounds write vulnerability can allow an attacker to modify the kernel's instructions residing in this shared space, fundamentally altering the program's intended logic.

Local Memory. Each thread possesses a private local memory space. This thread-local space stores function arguments, local variables, and control-flow data such as the return address. A stack-based buffer overflow, a prevalent type of memory error, can overwrite the saved return address. This vulnerability is the foundation for Return-Oriented Programming (ROP) attacks [10], allowing an attacker to hijack the thread's control flow and divert execution to malicious payloads.

The existence of these vulnerabilities highlights the need for robust mechanisms to detect and mitigate memory corruption in GPU kernels.

2.2 Deep Learning Operators

Deep learning (DL) frameworks like PyTorch [22] 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 [3], serve as the elemental building blocks for constructing neural networks. 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 [3], which are executed on the GPU.

Python Code

```
import torch
m1 = torch.randn(40, 40, 40).cuda()
model = torch.nn.AdaptiveAvgPool2d(
        output_size = [1, 67108607]).cuda()
model(m1)
```

CUDA Code

```
// Host Code
at::native::adaptive_avg_pool2d_out_cuda_template
int64_t osizeH = output_size[0];
int64_t osizeW = output_size[1];
output.resize_({sizeD, osizeH, osizeW});
adaptive_average_pool<<<...>>>(..., osizeH, osizeW, ...);

// Device Code at::native::adaptive_average_pool
__global__void adaptive_average_pool(..., scalar_t *output,
                             int osizeH, int osizeW, ...)
{  for(oh = ostartH; oh < oendH; oh += ostepH) {
      for(ow = ostartW; ow < oendW; ow += ostepW) {
           // Potential out-of-bounds calculation
           scalar_t *ptr_output = output + oh*osizeW + ow;
           // Accessing invalid memory
           *ptr_output = sum / kH / kW;
```

Fig. 2: From Python API to CUDA kernel.

As illustrated in Fig. 2, high-level Python operator calls are translated into low-level CUDA kernel executions, where subtle implementation bugs can lead to various memory errors.

The complexity arises from the vast parameter space of each operator. A single convolution operator [6], 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 [9]. This manual, performance-driven optimization often bypasses safer, high-level abstractions, making the kernel code a fertile ground for subtle memory errors [12] 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 [24] and autonomous driving [14], or be exploited for security attacks [21, 23].

State-of-the-art DL fuzzers focus on the compiler stack, generating valid neural networks to find bugs [15, 16]. This 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 Fig. 3, the system consists of three main phases: operator modeling, constraint-based test case generation, and cross-framework execution.

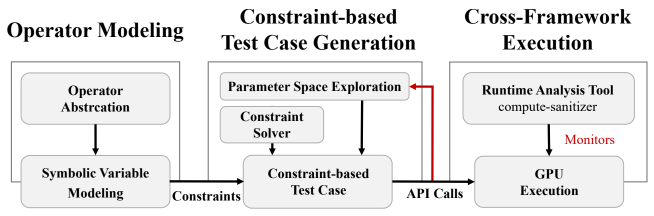

Operator Modeling
Constraint-based Test Case Generation
Cross-Framework Execution
Operator Abstraction
Symbolic Variable Modeling
Parameter Space Exploration
Constraint Solver
Constraint-based Test Case
Runtime Analysis Tool compute-sanitizer
Monitors
GPU Execution
Constraints
API Calls

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) is represented by a unified model that defines the interface for input/output shapes and parameter constraints.

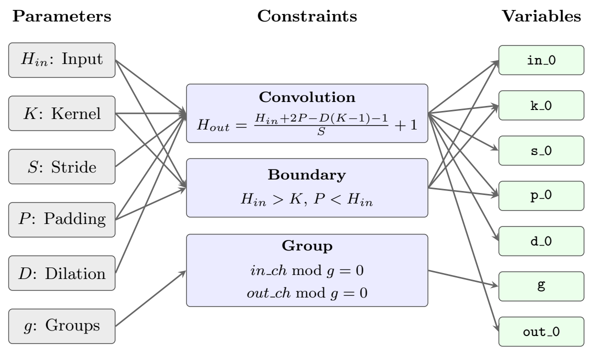


Fig. 4: Constraint modeling for convolution operators.

As illustrated in Fig. 4, GPU-Fuzz encodes operator semantics into a set of constraint formulas with symbolic variables. Take the convolution operator in Fig. 4 as an example, its core constraint formula [6] is $H_{out} = \frac{H_{in}+2P-D(K-1)-1}{S} + 1$, 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. Only after an input satisfies the constraint formulas, can the operator be correctly instantiated and executed. Aside from this core constraint, there are also various additional constraints (e.g., $H_{in} > K$) to ensure the semantic correctness of the operator. This constraint modeling approach allows us to effectively generate valid test cases for DL operators.

To ensure correctness, we manually extract the constraint formulas from the documentation of each operator. It takes approximately a month for two authors to extract the constraint formulas from the documentation of each operator independently and then cross-check their results. These two authors are experts in the field of deep learning and have a deep understanding of the semantics of each operator, which ensures the correctness of GPU-Fuzz to a great extent. In total, we extracted 45 constraints for 13 operators.

3.2 Constraint-based Test Case Generation

Constraint solving. Once an operator is modeled according to its constraints, GPU-Fuzz employs Z3's SMT solver [5] to find a satisfying assignment for all symbolic variables. As illustrated in Fig. 5, to generate a test case for a convolution operator, the solver must find concrete values for H_{out}, H_{in}, P, D, K, and S that satisfy the set of constraints shown in the figure. In the case of Fig. 5, a possible solution is $H_{out} = 126$, $H_{in} = 128$, $P = 1$, $D = 1$, $K = 5$, and $S = 1$.

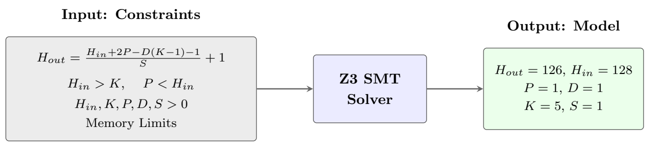


Fig. 5: Constraint solving process.

Although SMT solvers like Z3 [5] effectively find a solution for the given constraints, they are not designed to systematically explore the entire parameter space. Specifically, solvers like Z3 [5] tend to return a single boundary solution for the symbolic variables (e.g., $x = 0$ for $x \geq 0$), which is insufficient for a comprehensive fuzzing campaign. To address this limitation, previous works like NNSmith [15] force Z3 [5] to return multiple solutions by adding a periodic perturbation to the original constraint (e.g., $x > 2^n$). GPU-Fuzz, on the other hand, employs a novel constraint-guided search strategy to systematically explore the parameter space.

Parameter space exploration. To systematically explore the parameter space, GPU-Fuzz employs an iterative constraint-guided search strategy, as illustrated in Fig. 6. The process begins with an initial solution, $s1$ (e.g., $stride = 10, kernel\_size = 3$), found by the solver. At each iteration, the system randomly selects a parameter dimension to constrain. For instance, it might first select $stride$, adding constraints to exclude its current value, which guides the solver to a new solution like $s2$. In the next step, the system could randomly select $kernel\_size$, accumulating new constraints (e.g., $kernel\_size \neq 3$ and $h(kernel\_size) \neq h(3)$) upon the existing ones. This compels the solver to navigate towards unexplored regions, yielding another solution like $s3$.

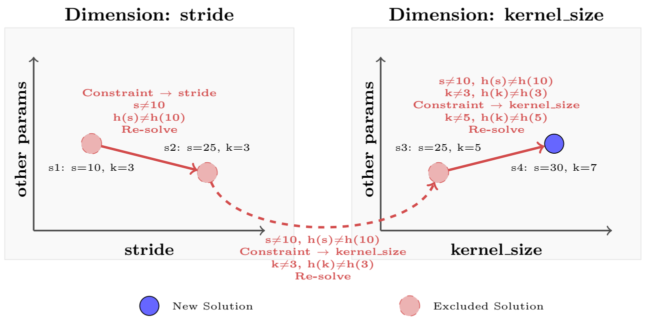
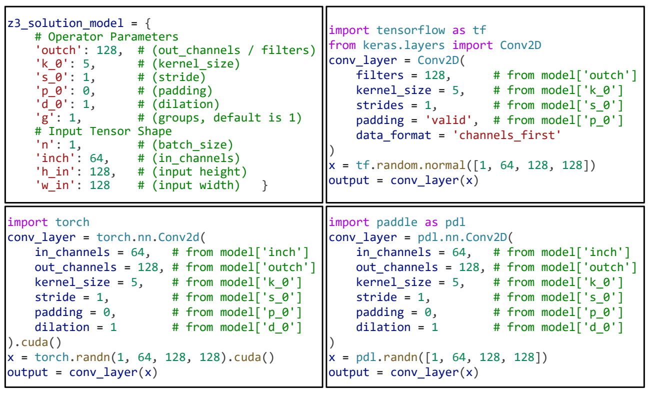


Fig. 6: An example of the iterative parameter space exploration. At each step, a parameter is randomly selected and a new constraint that excludes its current value is incrementally added to guide the search for the next solution.

Notably, to enhance both solver efficiency and solution diversity, we propose to incorporate not only direct value exclusion (e.g., $stride \neq 10$) but also hash-based constraints (e.g., $h(stride) \neq h(10)$). The hash function transforms input values through a series of bit-mixing operations that introduce dispersion properties. Specifically, the function applies alternating right-shift, XOR, and multiplication operations to ensure that even small changes in input values result in different hash values, effectively avoiding identity mapping and poor distribution. The hash-based constraint complements the direct value exclusion by preventing the solver from exploring similar regions, which significantly improves solution diversity.

This incremental strategy ensures that GPU-Fuzz can continuously and efficiently generate diverse test cases.

3.3 Cross-framework Execution



```
z3_solution_model = {
    # Operator Parameters
    'outch': 128,  # (out_channels / filters)
    'k_0': 5,      # (kernel_size)
    's_0': 1,      # (stride)
    'p_0': 0,      # (padding)
    'd_0': 1,      # (dilation)
    'g': 1,        # (groups, default is 1)
    # Input Tensor Shape
    'n': 1,        # (batch_size)
    'inch': 64,    # (in_channels)
    'h_in': 128,   # (input height)
    'w_in': 128    # (input width)   }
```

```
import tensorflow as tf
from keras.layers import Conv2D
conv_layer = Conv2D(
    filters = 128,      # from model['outch']
    kernel_size = 5,    # from model['k_0']
    strides = 1,        # from model['s_0']
    padding = 'valid',  # from model['p_0']
    data_format = 'channels_first'
)
x = tf.random.normal([1, 64, 128, 128])
output = conv_layer(x)
```

```
import torch
conv_layer = torch.nn.Conv2d(
    in_channels = 64,   # from model['inch']
    out_channels = 128, # from model['outch']
    kernel_size = 5,    # from model['k_0']
    stride = 1,         # from model['s_0']
    padding = 0,        # from model['p_0']
    dilation = 1        # from model['d_0']
).cuda()
x = torch.randn(1, 64, 128, 128).cuda()
output = conv_layer(x)
```

```
import paddle as pdl
conv_layer = pdl.nn.Conv2D(
    in_channels = 64,   # from model['inch']
    out_channels = 128, # from model['outch']
    kernel_size = 5,    # from model['k_0']
    stride = 1,         # from model['s_0']
    padding = 0,        # from model['p_0']
    dilation = 1        # from model['d_0']
)
x = pdl.randn([1, 64, 128, 128])
output = conv_layer(x)
```

Fig. 7: Cross-framework materialization example.

Once a valid set of parameters is generated, GPU-Fuzz executes it as a test case across multiple deep learning frameworks,

Tab. 1: Supported Operators 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 (1d, 2d, 3d), AdaptiveAvgPool (1d, 2d, 3d), AdaptiveMaxPool (1d, 2d, 3d)
Padding	ReflectionPad (1d, 2d, 3d), ReplicationPad (1d, 2d, 3d), ZeroPad (1d, 2d, 3d), ConstantPad (1d, 2d, 3d), CircularPad (1d, 2d, 3d)
Element-Wise Unary	Activation: ELU, ReLU, GELU, Sigmoid, Tanh, ... Arithmetic: abs, sin, cos, sqrt, exp, log, ...
Element-Wise Binary	add, sub, mul, div, pow, remainder, logaddexp, atan2, ...
Matrix Ops	MatMul, BMM
Concat	cat, concat

including PyTorch [22], PaddlePaddle [17], and TensorFlow [1], to detect GPU memory bugs. As illustrated in Fig. 7, this process translates the abstract, framework-agnostic parameters from the solver into concrete parameters with specific API calls. For instance, the generic `outch` parameter is mapped to `out_channels` in PyTorch and `filters` in TensorFlow. To detect memory errors and kernel failures, each execution is wrapped by NVIDIA's compute-sanitizer [19]. This approach significantly improves the detection of memory errors.

4 IMPLEMENTATION

This section details the implementation of the system design described in Section 3. Our system is developed in Python and comprises 2,628 lines of code.

We implemented a library where each operator is represented as a distinct class to realize the operator modeling and constraint-based test case generation concepts described in Section 3. This modeling approach was applied to 13 operators (Table 1), chosen for their prevalence in deep learning models and their complex, error-prone memory access patterns.

The translation process (Section 3.3) is implemented to translate the generated parameter values into framework-specific API calls. When compute-sanitizer [19] detects an error during execution, the system automatically archives the execution logs 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 DL fuzzers in terms of test case generation and bug discovery, 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	
Operating System	Ubuntu 24.04.2 LTS
NVIDIA Driver	580.82.07
CUDA Runtime	12.8.93
Python	3.11.13
Frameworks	
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 7 distinct memory access violations (e.g., out-of-bounds or misaligned writes). Among these, 5 were silent memory corruptions that do not trigger any API-level crash and are only detectable with specialized tools like compute-sanitizer [19].
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.

5.3 Comparative Study

To quantitatively validate our approach, we conducted a comparative study against NNSmith [15], a state-of-the-art DL 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 $19{,}063 \pm 360$ test cases and uncovered 296 ± 19 bugs. Most of its findings are numerical mismatches rather than memory-safety issues.

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

ID	Framework	Operator	Bug Type	Root Cause	Failure Mode	Status
Bug$_1$	PyTorch	`conv_transpose2d`	OOB Global Write	Incorrect grid dimension calculation	GPU-Level Exception (CUBLAS)	Confirmed
Bug$_2$	PyTorch	`bmm_sparse`	Misaligned Global Write	Incorrect pointer arithmetic in CUSPARSE	Silent Memory Corruption	Reported
Bug$_3$	PyTorch	`adaptive_avg_pool2d`	OOB Global Write	Flawed boundary checks in CUDA kernel	Silent Memory Corruption	Confirmed
Bug$_4$	PyTorch	`replication_pad2d`	OOB Global Write	Incorrect grid dimension calculation	Silent Memory Corruption	Confirmed
Bug$_5$	PyTorch	`adaptive_max_pool2d`	OOB Global Write	Flawed boundary checks in CUDA kernel	Silent Memory Corruption	Confirmed
Bug$_6$	PyTorch	`conv_transpose3d`	OOB Shared Read	Incorrect index calculation for shared memory	Silent Memory Corruption	Fixed
Bug$_7$	PyTorch	`reflection_pad1d`	Invalid Launch Config	Integer overflow in `torch.compile` symint logic	GPU-Level Exception (CUDA)	Confirmed
Bug$_8$	TensorFlow	`Conv2D`	OOB Global Read	Incorrect index calculation in kernel	Silent Read / Downstream Crash	Confirmed
Bug$_9$	TensorFlow	`Conv2D`	Integer Overflow	Overflow in launch config calculation	CPU-Side Assert	Confirmed
Bug$_{10}$	PaddlePaddle	`conv2d_transpose`	Precondition Violation	Integer overflow in tensor dimension calculation	CPU-Side Assert	Confirmed
Bug$_{11}$	PaddlePaddle	`conv3d_transpose`	Illegal Instruction	Invalid parameters passed to cuDNN kernel	GPU-Level Exception (cuDNN)	Confirmed
Bug$_{12}$	PaddlePaddle	`conv2d_transpose`	Bad API Parameter	Invalid parameter combination passed to cuDNN	GPU-Level Exception (cuDNN)	Confirmed
Bug$_{13}$	PaddlePaddle	`conv2d_transpose`	Invalid Launch Config	Incorrect grid/block dimension calculation	GPU-Level Exception (CUDA)	Confirmed

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 80 ± 7 configuration errors. These memory errors represent severe security vulnerabilities that could result in data corruption, information leakage, or system crashes.

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	80 ± 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.

Python Code

```python
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)
```

CUDA Code

```cpp
// C++ Code at::native::...::slow_conv_transpose2d
int64_t n_elements_64 =calculate_total_elements(...);
int n_elements_32 = (int)n_elements_64; // Integer Overflow
dim3 grid =calculate_grid(n_elements_32);
col2im_kernel<<<grid, block, ...>>>(...);

// CUDA Device Code at::native::col2im_kernel
__global__voidcol2im_kernel(..., float*output)
{
   int64_t index =blockIdx.x *blockDim.x+threadIdx.x;
   output[index] = ...; // Accessing invalid memory
```

Fig. 8: A minimal PoC in Python and the corresponding CUDA implementation that triggers a memory bug in PyTorch's `ConvTranspose2d`.

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, GPU-Fuzz uncovered 26 ± 5 memory errors that pose 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 blind spot in GPU memory security testing that existing DL fuzzers frequently miss. The two approaches are complementary: NNSmith excels at uncovering compiler-related bugs and numerical inconsistencies, 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 Fig. 8.

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) combined with input dimensions of (10, 40000, 2). While these values are semantically valid according to PyTorch's API, they represent a corner case that is unlikely to be covered by manual tests. As illustrated in Fig. 8, the root cause is an integer overflow in the C++ host code: when calculating the total number of elements for the CUDA kernel, a 64-bit integer value is cast to a 32-bit integer, which causes truncation. This overflowed value is then used to calculate the CUDA grid dimensions, resulting in an undersized grid that cannot cover all required memory operations. When the `col2im_kernel` executes, threads calculate 64-bit indices that exceed the actual allocated buffer size, leading to out-of-bounds memory writes.

```
========= 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. 9: The error message for the PoC.

As shown in Fig. 9, compute-sanitizer detects that a CUDA thread attempts to write to an out-of-bounds address, which is before the nearest valid allocation. This type of silent memory corruption is a severe bug that can lead to incorrect results or unpredictable behavior without causing an explicit Python crash, and 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 13 operators, 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

Fuzzing for DL systems has received continuous attention over the past years [25]. To capture bugs from complicated modern DL compiler stacks, one of the effective approaches is generating valid neural network models. Works like NNSmith [15], TVMFuzz [25], and HirGen [16] pioneered this direction, proving effective at identifying compiler-related bugs, such as graph-level optimization issues and IR transformation errors. Following this, other works propose testing different components of the DL stack. LEMON [26], for example, tests DL library implementations by generating model variants to detect inconsistencies across different libraries. More recent studies, like Orion [11], specifically focus on the API layer by generating test inputs guided by historical bug patterns. While effective, these approaches primarily operate at higher abstraction levels. They are not designed to systematically probe the low-level memory access patterns within the GPU kernels that execute the operators. Testing GPU kernels, in general, has its own line of research. GPUVerify [2] employs formal methods to verify kernel correctness by detecting data races and barrier divergence. DeepSmith [4] generates random CUDA programs from scratch to stress-test the CUDA compiler [9] itself.

In contrast, GPU-Fuzz differs from this previous work. Instead of generating entire models like NNSmith [15] or entire CUDA programs like DeepSmith [4], it focuses on parameter space fuzzing of existing DL operators to uncover bugs at the kernel level.

8 RESPONSIBLE DISCLOSURE

We have disclosed all 13 discovered bugs responsibly to the respective development teams of PyTorch [22], TensorFlow [1], and PaddlePaddle [17] 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. We are committed to collaborating with the framework maintainers to help improve the security and robustness of the deep learning ecosystem.

9 CONCLUSION

We introduced GPU-Fuzz, a constraint-guided fuzzer that aims to find memory errors in deep learning operators. By shifting the focus from network models to the operator parameter space, 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 model and operator parameter space 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] 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.

[4] 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.

[5] 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.

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

[7] 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.

[8] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, 2005.

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

[10] Yanan Guo, Zhenkai Zhang, and Jun Yang. GPU memory exploitation for fun and profit. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 4033–4050, Philadelphia, PA, August 2024. USENIX Association. ISBN 978-1-939133-44-1. URL https://www.usenix.org/conference/usenixsecurity24/presentation/guo-yanan.

[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] 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.

[13] David B Kirk and W Hwu Wen-Mei. *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann, 2016.

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

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.