THE CHIPS TO SYSTEMS CONFERENCE

SPONSORED BY: CEDA IEEE Council on Electronic Design Automation, SIGDA ACM

GPU-Fuzz

Finding memory errors in deep learning frameworks

Zihao Li, Hongyi Lu, Yanan Guo, Zhenkai Zhang,
Shuai Wang, Fengwei Zhang

SUSTech Southern University of Science and Technology
COMPASS Lab COMPuter And System Security Lab
HKUST
University of Rochester
CLEMSON UNIVERSITY

Background: why GPU memory errors matter

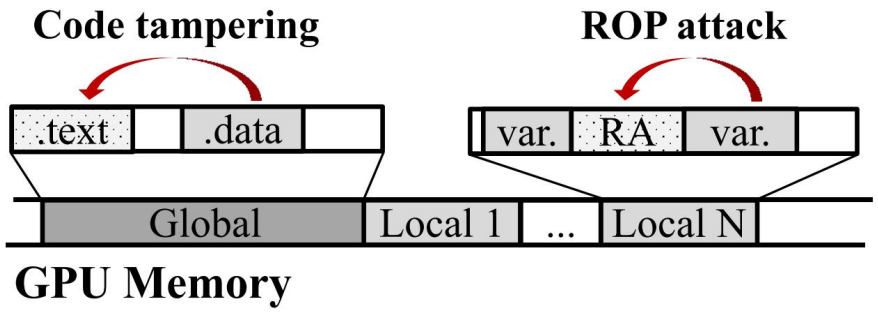

Fig. 1: GPU memory layout and potential attacks.

- **Code tampering**: modifying code through memory errors.
- **ROP attack**: memory errors can redirect control flow.

- DL frameworks rely on **hand-written CUDA/C++ kernels**.
- Kernel code involves pointer arithmetic, thread indexing, and shared/global memory.
- Memory errors may cause:
  - silent data corruption
  - GPU exceptions
  - out-of-bounds writes
  - control-flow or code-region corruption
- These are reliability and security issues, not just crashes.

Problem: blind spots in existing DL fuzzers

Table 1. Comparison Between Fuzz4CUDA, CuFuzz, and GPUFuzz

Feature	Fuzz4CUDA	CuFuzz	GPUFuzz (Ours)
Target	NVIDIA Libraries (e.g., cuBLAS)	General GPU Apps (e.g., Rodinia)	**DL Frameworks (e.g., PyTorch)**
Methodology (Fuzzer Engine)	Random Mutation (AFL++)	Random Mutation (AFL++)	**Constraint Solving (SMT Solver)**
Mathematical Structure Aware	No	No	**Yes**
Execution	GPU	Translated to CPU	**GPU**

- Existing fuzzers mostly test **model structures** or **general CUDA APIs**.
- GPU memory bugs often depend on **operator parameters**:
  - shape
  - stride / padding / dilation
  - groups
  - dtype and layout
- Model-level fuzzing misses many kernel-level memory bugs.

Key insight: bugs hide in the operator parameter space

- GPU-Fuzz focuses on **how operator parameters determine GPU kernel memory access**.
- For **convolution-like** operators, shape, kernel, stride, padding, dilation, and groups interact.
- **API-valid does not mean kernel-safe.**
- So testing should include operator-level parameter fuzzing, not only model-level fuzzing.

Python Code

```python
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.

Overview: GPU-Fuzz architecture

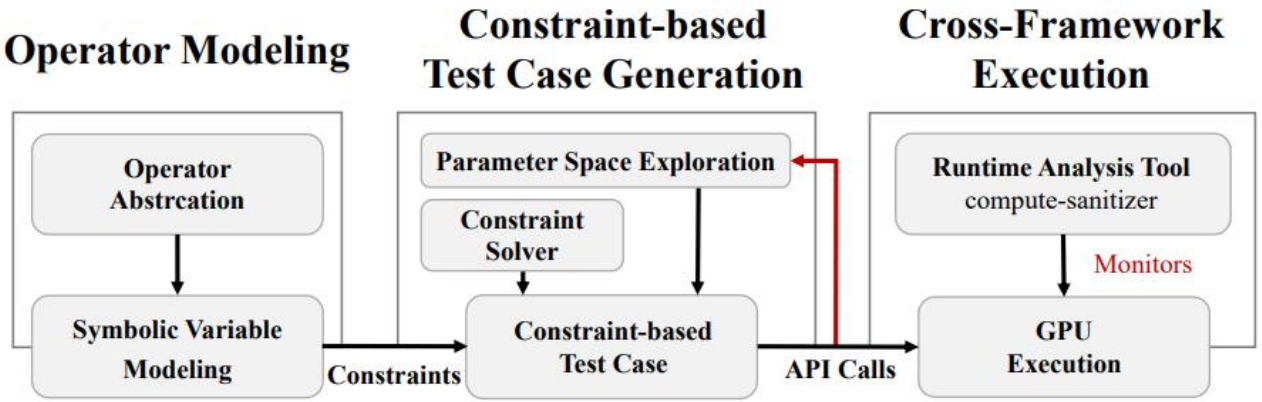

Fig. 3: The architecture of the GPU-Fuzz system.

- Three stages:
  - **Operator Modeling**: formalize operator semantics and boundary conditions as constraints
  - **Constraint-based Test Case Generation**: use Z3 to generate valid parameter combinations
  - **Cross-framework Execution**: run generated tests under compute-sanitizer
- Main advantages:
  - API-valid tests
  - systematic boundary exploration
  - shared models across PyTorch, TensorFlow, and PaddlePaddle

Operator Modeling: turning parameters into constraints

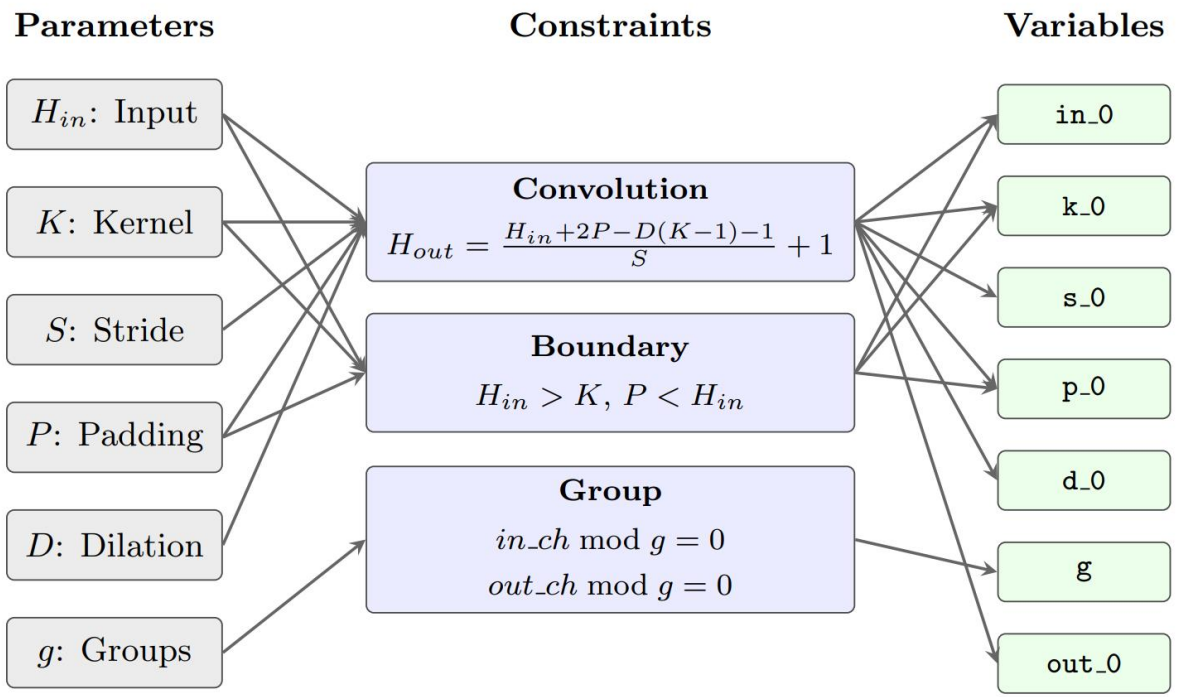

Fig. 4: Constraint modeling for convolution operators.

- Each operator type is abstracted as a parameter model.
- The model includes:
  - input/output shape relations
  - parameter validity constraints
  - mappings to framework APIs
- For convolution, GPU-Fuzz models the core shape formula and adds constraints such as:

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

  - Hin > K
  - P < Hin
  - in_ch mod group = 0
  - out_ch mod group = 0
- The paper extracts **45 constraints for 13 operators**.

Constraint-guided Parameter Exploration: valid and efficient generation

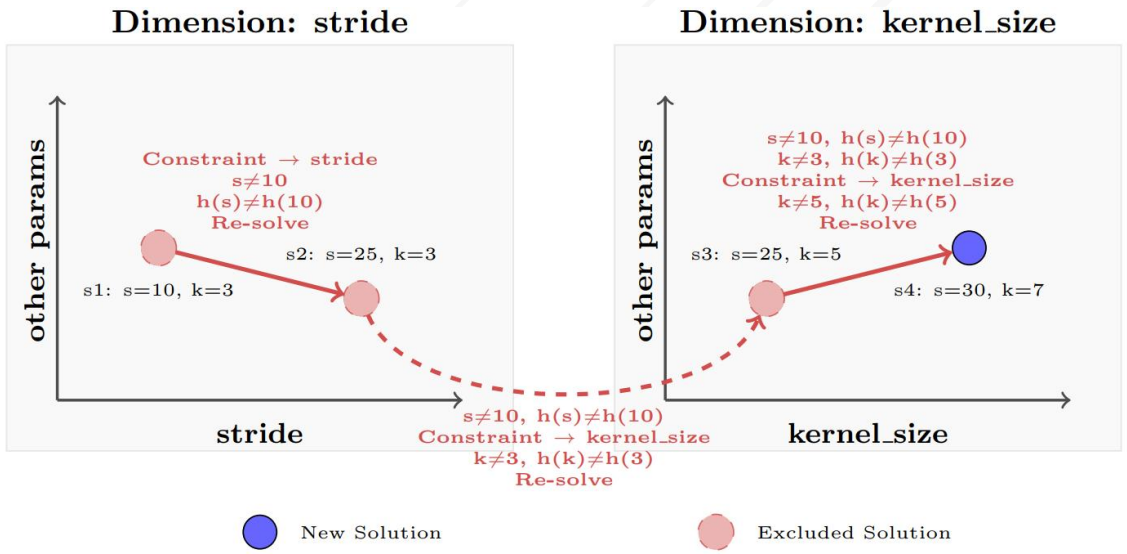

Fig. 6: An example of the iterative parameter space exploration. At each step, a parameter is randomly selected and a new constraint that excludes its current value is incrementally added to guide the search for the next solution.

- Z3 generates **valid** operator parameters.
- **Iterative constraints improve diversity**:
  - generate one solution
  - **choose one parameter dimension randomly**
  - add direct exclusion, such as **stride != 10**
  - add a hash-based constraint, such as **h(stride) != h(10)**
  - push the solver into a **new** parameter region

- ConvTranspose2d overflow:
  - **AFL++**: 1,093.2 executions
  - **GPU-Fuzz**: 21.8 executions

Cross-framework Execution: from abstract parameters to real APIs

- Generate **framework-independent** parameters.
- Map them to concrete APIs:
  - **PyTorch**: out_channels
  - **TensorFlow**: filters
  - **PaddlePaddle**: corresponding names
- Run each test under NVIDIA compute-sanitizer.
- Detect OOB access, misalignment, invalid access, and kernel exceptions.

```python
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)   }
```

```python
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)
```

```python
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)
```

```python
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.

Results: 13 previously unknown bugs

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

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

- Found across PyTorch, TensorFlow, and PaddlePaddle.
- Affected **common operators**: convolution, pooling, and padding.
- Some bugs caused **silent memory corruption**.
- Root causes:
  - **Geometric boundaries**: extreme valid shapes break kernel mapping or scheduling.
  - **Numeric boundaries**: oversized parameters overflow counters, offsets, or launch sizes

Comparison: GPU-Fuzz vs. NNSmith

- **Setup**:
  - PyTorch target
  - same hardware
  - 5 runs per tool
  - 4 GPU-hours per run
- **Results**:
  - **NNSmith**: 19,063 tests; mainly numerical inconsistencies
  - **GPU-Fuzz**: 51,860 tests; 26 critical memory errors + 80 configuration errors
- **Conclusion**:
  - **NNSmith** targets model/compiler bugs.
  - **GPU-Fuzz** targets operator/kernel memory safety.

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.

Case Study and Summary

- **Case study**: PyTorch ConvTranspose2d
- **Triggering parameters**:
  - input shape: (10, 40000, 2)
  - stride: (200, 200)
- **Root cause**:
  - 64-bit element count is **cast** to 32-bit.
  - Integer overflow corrupts CUDA grid configuration.
  - Kernel threads exceed the real buffer and cause OOB writes.
- **Summary**:
  - GPU-Fuzz shifts fuzzing to operator parameters.
  - SMT solving keeps tests valid and boundary-focused.
  - It finds memory bugs **missed** by existing DL fuzzers.
- **Limitations**:
  - manual operator modeling
  - compute-sanitizer focuses on memory errors

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

```
// 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`.

```
========= 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.

THE CHIPS TO SYSTEMS CONFERENCE

Thank you

GPU-Fuzz: Finding Memory Errors in Deep Learning Frameworks

Zihao Li

Email: lizh2025@mail.sustech.edu.cn