Artifact Evaluated
NDSS
Available
Functional
Reproduced

FuzzWarp: GPU-aware Memory Error Detection in Deep Learning Frameworks

Georgios Kokolakis
Georgia Institute of Technology
gkokolakis6@gatech.edu

Neophytos Christou
Brown University
neophytos_christou@brown.edu

Mohamed Tarek Ibn Ziad
NVIDIA
mtarek@nvidia.com

Vaggelis Atlidakis
University of Athens
vatlidak@di.uoa.gr

Angelos D. Keromytis
Georgia Institute of Technology
angelos@gatech.edu

Vasileios P. Kemerlis
Brown University
vpk@cs.brown.edu

***Abstract*—We present FuzzWarp, a GPU-aware fuzzer for detecting memory-safety violations in GPU kernels of Deep Learning (DL) frameworks. Unlike existing fuzzers for DL code, which predominantly target high-level Python APIs—and hence may miss GPU-specific execution paths, silently trigger CPU fallback behavior, or lack effective oracles for detecting GPU memory errors—, FuzzWarp takes a bottom-up approach that directly targets low-level GPU kernels. More specifically, it obtains precise control over device dispatch and mutates kernel inputs with dimension-product overflows, boundary values, and zero-dimension tensors.**

FuzzWarp integrates three mechanisms into a DL framework's host-to-GPU execution path: (1) dispatch verification, which confirms that each input reaches the intended GPU kernel before fuzzing it (29–44% of inputs from existing fuzzers do not); (2) device-aware allocation, which creates each mutated tensor through the framework's allocator on the device the kernel expects; and (3) asynchronous fault attribution, which synchronizes the GPU after every mutation and checks device memory accesses with NVIDIA's Compute Sanitizer, so that each fault is attributed to the mutation that caused it.

FuzzWarp synthesizes Proof-of-Vulnerability (PoV) snippets that lift low-level kernel errors back to reproducible high-level Python API calls. It consistently outperformed state-of-the-art DL fuzzers across multiple TensorFlow versions, and uncovered 99 previously unknown GPU-specific memory-safety bugs in TensorFlow and PyTorch (of which 62 are acknowledged and 37 have been fixed or have approved fixes already). Finally, we validate an end-to-end scenario remotely triggering one of FuzzWarp's uncovered bugs by submitting a crafted, oversized request to a vulnerable model served by the NVIDIA Triton Inference Server.

I. Introduction

Artificial intelligence (AI), and Deep Learning (DL) in particular, are increasingly used in critical infrastructure across numerous domains. Modern diagnostic pipelines in healthcare [1], perception modules in autonomous vehicles [2], personalized recommendation systems used for online advertisement [3], and large-scale natural-language models [4], are just a few examples where DL is applied. Since these applications typically rely on using DL models with billions of parameters at training, as well as during inference/production time, they inevitably utilize GPUs, which excel in handling large-scale matrix operations in a massively parallel fashion.

DL frameworks, such as TensorFlow [5] (developed by Google) and PyTorch [6], have become the canonical layer of abstraction atop which modern applications are built and operate. To help developers easily define and seamlessly integrate DL models into the desired application pipelines, these frameworks expose high(er)-level APIs and abstract away (from application and model developers) the low(er)-level details of how support for efficient execution on heterogeneous GPU-accelerated hardware is implemented. In practice, the essential parts of DL-specific operations (called *kernels*) are implemented in memory-unsafe languages, like C/C++, and typically utilize the CUDA framework [7]: the de-facto parallel computing platform, and API library, for general-purpose computing on NVIDIA GPUs.

GPUs share common memory-safety [8], [9] and side-channel [10], [11] vulnerabilities with CPUs, but their deeply parallel architectures and specialized memory hierarchies allow for additional GPU-specific attack vectors. Recent work by Guo et al. [12] argues that out-of-bounds memory accesses can be exploited to access data beyond their legitimate scope and code-reuse attacks can be leveraged against CUDA programs. These attacks extend to DL deployments. Roh et al. [13] use memory-safety bugs in the GPU kernels of DL frameworks to corrupt host process memory through the unified CPU and GPU address space, leading to privilege escalation. He et al. [14] show that such bugs can be exploited during LLM inference, and Roels et al. [15] exploit GPU memory errors remotely during ML inference. Several mitigation strategies have been proposed against such threats, ranging from fine-grained memory isolation mechanisms [16]–[18] to compiler-inserted runtime instrumentation approaches [19], [20], as well as generic program shepherding-like tools [21], [22]. For example, Guardian [17], cuCatch [19], and CuSafe [20] report average runtime overheads of 9%, 19%, and 13%, respectively.

Network and Distributed System Security (NDSS) Symposium 2027
22–26 March 2027, Seoul, Republic of Korea
ISBN 978-1-970672-09-1
https://dx.doi.org/10.14722/ndss.2027.230306
www.ndss-symposium.org

Complementary to these mitigations, *program testing* approaches, such as those using fuzz testing to automatically uncover memory-safety errors in DL frameworks [23], have primarily focused on discovering CPU-specific memory errors, frequently overlooking GPU-specific ones—or lack robust detection mechanisms for GPU memory errors. We assessed 22 state-of-the-art DL-framework fuzzers and identified three common classes of shortcomings: (i) lack of detection methods tailored to the asynchronous and silent nature of GPU memory errors; (ii) mutation strategies that do not account for edge cases specific to GPU code; and (iii) no device-dispatch verification, allowing 29–44% of generated inputs in our baseline campaigns to execute another GPU kernel or helper, fall back to the CPU, or fail before dispatch (§IX-B).

Lastly, while generic GPU-kernel fuzzers exist [24], [25], they cannot be applied to DL frameworks "as-is". DL frameworks perform substantial host-side *preprocessing* before launching a computation in the GPU—e.g., validating shapes/`dtypes`, selecting backend-specific kernel variants/-implementations, and configuring launch parameters. This can reject mutated inputs early, or route execution to a different backend, meaning that a standalone GPU-kernel harness often fails to reproduce the same host-GPU interactions.

To address the aforementioned gaps, we present `FuzzWarp`: a GPU-aware fuzzing system for DL frameworks. `FuzzWarp` uses kernel instrumentation to expose GPU kernel entry points, boundary-value mutation to generate tensor inputs, and sanitizer-based checking to validate memory errors. Some prior fuzzers have used these mechanisms separately. `FuzzWarp` integrates them into the framework's host-to-GPU execution path. It uses framework tests to reach the instrumented GPU kernels and verifies the intended GPU implementation before fuzzing its inputs. Because mutation occurs after device placement, `FuzzWarp` allocates mutated tensors through the framework's allocator on the device selected for each input. Moreover, it synchronizes GPU execution after each invocation to attribute deferred faults to the triggering mutation. Most importantly, `FuzzWarp` maps each fault to a PoV that uses public Python APIs. `FuzzWarp`'s contribution in the design space is the end-to-end integration of the above mechanisms with dispatch verification, device-aware allocation, asynchronous fault attribution, and PoV synthesis.

Using `FuzzWarp`, we identified 99 previously unknown GPU-specific memory errors: 30 in TensorFlow v2.20, all confirmed by the TensorFlow developers, and 69 in PyTorch v2.13. In summary, we make the following contributions:

- *GPU-aware fuzzer for DL frameworks.* We present the design and implementation of `FuzzWarp`, a fuzzer specifically designed to systematically detect GPU memory errors in DL frameworks. Unlike existing approaches, `FuzzWarp` employs (a) dispatch verification, (b) device-aware allocation, and (c) asynchronous fault attribution to comprehensively target GPU-specific vulnerabilities.
- *Analysis of the DL fuzzing landscape.* We provide a taxonomy of 22 state-of-the-art DL fuzzers based on their detection capabilities, revealing that only 2 of them detect GPU-specific errors (7 errors in total).
- *Empirical study of GPU kernels in DL frameworks.* `FuzzWarp` discovered 99 previously unknown GPU-specific memory errors: 30 in TensorFlow v2.20 and 69 in PyTorch v2.13 (§IX-C). We analyze and categorize these errors. We further demonstrate that a `FuzzWarp`-identified TensorFlow vulnerability can be triggered remotely during model inference, through a single valid JPEG image submitted to an unmodified VGG-11 model served by NVIDIA Triton [26] (§IX-E).

We reported the bugs following each project's disclosure guidelines; 62 have been acknowledged so far. The TensorFlow developers acknowledged all 30, and 28 of them have merged or approved fixes. For PyTorch, we reported only bugs whose PoVs use APIs considered *public*. At the time of writing, 32 of the 69 PyTorch bugs have been acknowledged and 9 fixed; 22 of the acknowledged bugs were grouped under a single umbrella issue on missing shape checks. No CVEs have been assigned so far. These findings point to security risks in production DL frameworks that existing fuzzers miss.

II. Background

A. DL Framework Architecture

DL frameworks, such as TensorFlow [5] and PyTorch [6], consist of multiple *operations* [5]: i.e., fundamental units of computation that perform mathematical transformations on input tensors (e.g., tensor addition, convolution ops). Operations are typically used via *publicly-exposed, high-level* language APIs (e.g., Python APIs), but their underlying implementations, called *kernels* [5], are written in low-level languages (e.g., C++, hand-tuned assembly). These two layers are connected via language *bindings*, which expose the low-level kernels to the high-level language (typically Python) intended for use by the developers and are usually wrapped by high(er)-level APIs to abstract away implementation details.

Note that the term "kernel" is *overloaded*. In DL frameworks (e.g., TensorFlow), a kernel refers to the implementation of an operation (e.g., a TensorFlow `(Async)OpKernel` method and its surrounding native code), whereas in GPU programming platforms (e.g., CUDA), a kernel denotes GPU-device code executed by the accelerator. Unless stated otherwise, we use the term *kernel* to refer to the former and the term *CUDA kernel* to refer to the latter.

Kernels. To ensure performant execution, DL framework operations are implemented in low-level functions called *kernels*. Kernel code is designed to be *stateless* and self-contained to enable safe (and massive) parallelism. Kernel implementations are usually tied to a specific hardware/platform, and hence one operation can have multiple kernel *variants* for different hardware (e.g., CPUs, GPUs, and TPUs). Device-specific variants can be implemented either as standalone kernels, or as specializations of the same kernel prototype (e.g., using C++ function templating [27]), with common code that validates the input arguments and dispatches (kernel) execution to a device-specific implementation [28], [29].

```
1  // Vulnerable kernel: missing bounds check
2  __global__ void vuln_kernel(float *data, int N) {
3    // Calculate global thread ID
4    int idx = blockIdx.x * blockDim.x + threadIdx.x;
5
6    // If N is not a multiple of blockDim.x,
7    // the last block accesses OOB memory
8    data[idx] = data[idx] * 2.0f;
9  }
10
11 int main(void) {
12   int N = 100;
13   float *d_data;
14   cudaMalloc(&d_data, N * sizeof(float));
15
16   // Launching 4 blocks of 32 threads
17   // (128 threads total). Threads 100-127
18   // trigger CUDA_ERROR_ILLEGAL_ADDRESS
19   vuln_kernel<<<4, 32>>>(d_data, N);
20   ...
21   // The memory error is reported here (much
22   // later from where the OOB takes place)
23   cudaDeviceSynchronize();
24   ...
25   return 0;
26 }
```

Listing 1: Example of a memory-unsafe CUDA kernel leading to asynchronous OOB access.

When an operation with multiple kernel implementations is invoked, the framework decides which hardware-specific variant should be executed based on various conditions. For example, to decide whether a kernel should be dispatched to its GPU-specific implementation (typically built atop CUDA [29] or ROCm [30]), DL frameworks first check *device availability*, to verify GPU hardware presence and driver accessibility, via runtime queries (e.g., `torch.cuda.is_available()`). *User-specified directives* allow explicit device placement through context managers, like `tf.device()` or `torch.device()`, overriding automatic placement logic. Finally, *argument data-type compatibility* validates that specific tensor data types are supported by the target GPU kernel [29], as many kernel implementations support specific precision types only (e.g., `float32`, `float16`).

Bindings. The low-level kernel implementations are (usually) not publicly exposed to developers (i.e., they cannot be directly invoked). To give developers access to the native kernels, DL frameworks provide automatically-generated bindings in high(er)-level languages (e.g., Python) [29], [31]. These bindings translate/(un)marshal the arguments/types of high-level APIs to the respective low-level (e.g., C++) ones, and dispatch execution to one of the available kernel implementations for the invoked operation.

High-level APIs and Models. Even though the high-level language bindings are developer-accessible, frameworks further provide other "higher-level" APIs, which are either thin wrappers around the raw operation bindings, or combine multiple bindings in order to construct more complex operations. These high(er)-level APIs are what developers typically use when building models [32].

B. GPU Memory Architecture

Modern GPUs expose a multi-tier memory hierarchy that is designed to achieve high performance via massive thread-level parallelism [33]. *Global memory* (DRAM) is usually large and uniformly-addressable by all device threads, but incurs high access latency; code and bulk data chunks primarily reside in global memory. *Shared memory*, physically "close" to each streaming multiprocessor (SM), is scoped to a thread block and is used for data reuse across threads and inter-thread communication. *Local memory* is used to spill per-thread temporaries that do not fit in registers; because it maps to global DRAM, excessive spilling degrades throughput and increases exposure to out-of-bounds memory accesses when index arithmetic is buggy. *Registers* reside in the SM register file, provide single-cycle access, and are private to each thread; exhausting them triggers spilling to local memory.

C. Memory Allocation in DL Frameworks

Whenever a tensor is created using a high-level API, DL frameworks use *internal memory allocators* in order to allocate memory for the tensor on the right *device* and of the proper *type* (§II-B) [5]. The *host allocator* manages the system's memory, which is typically used for tensors and operations that execute on the CPU. The *pinned host allocator* handles page-locked host memory, which enables high-throughput and asynchronous data transfers between the host and accelerator devices. In addition, each accelerator maintains a *device-specific allocator* (e.g., for GPU or TPU memory) that implements caching or best-fit policies to reduce fragmentation and reuse device memory efficiently.

The choice of allocator for each tensor argument is critical, as (a) pre-allocating tensors on the correct device minimizes cross-device data transfers (e.g., between CPU and GPU) and (b) not all types of tensors can reside on every memory type or device. For example, certain data types or objects are constrained to specific memory domains (e.g., string tensors in TensorFlow can only reside in host memory, even if the kernel that operates on them is executing on a GPU). Attempting to place such tensors on the wrong device memory will lead to runtime exceptions/faults.

D. GPU Memory Errors

GPU programs exhibit memory-safety violations similar to those that execute on CPUs: GPU programming frameworks, such as CUDA and OpenCL, are based on C/C++ and are hence inherently memory-unsafe. For instance, Listing 1 illustrates a common error where a CUDA kernel thread calculates a global memory offset based on its thread ID (ln. 4), but fails to verify that the index falls within object boundaries before accessing memory (ln. 8). Prior work [12], [34] has demonstrated that such code suffers from memory errors (e.g., buffer overflows [35]), which can be exploited using traditional techniques (i.e., code reuse [34], [36] or code injection [12], [37]) and lead to unauthorized data exfiltration, silent data corruption in ML models, or even privilege escalation [13].

TABLE I
BUG DETECTION IN TENSORFLOW BY EXISTING DL FUZZERS.

Oracles: † = detection of crashes and exceptions; ‡ = differential testing; ◊ = detection of semantic violations. Only 2 (of the 22) tools report GPU-specific memory errors, despite the widespread GPU usage in DL frameworks.

Fuzzer	GPU	CPU			Total	Oracles
	Mem. Errors	SIGSEGV	SIGFPE	SIGABRT		
ACETest [40]	**6**	35	11	85	137	†◊
IvySyn [23]	0	26	4	56	86	†
CEDAR [41]	0	11	11	48	70	†◊
DocTer [42]	0	9	12	39	60	†◊
Orion [43]	0	15	1	13	29	†‡
TensorScope [44]	0	13	0	16	29	†‡
SkipFuzz [45]	0	6	1	16	23	†
DFUZZ [46]	0	2	0	21	23	†‡
PathFinder [47]	0	2	0	16	18	†
TitanFuzz [48]	**1**	1	0	3	5	†‡
DEEPCONSTR [49]	0	0	0	4	4	†‡
FreeFuzz [50]	0	0	2	1	3	†‡
NablaFuzz [51]	0	0	1	2	3	†‡
DeepREL [52]	0	0	1	0	1	†‡
NeuRI [53]	0	0	0	1	1	†‡
ConFL [54]	-	-	-	-	-	†◊
EAGLE [55]	0	0	0	0	0	‡
MoCo [56]	0	0	0	0	0	†‡
LEMON [57]	0	0	0	0	0	†‡
Muffin [58]	0	0	0	0	0	†‡
FuzzGPT [59]	0	0	0	0	0	†‡
DLLens [60]	0	0	0	0	0	†‡

In contrast to CPUs, which typically raise a *synchronous* segmentation fault (`SIGSEGV`) at the exact instruction that references an invalid address, GPU memory faults often manifest *asynchronously*. A kernel may complete—or appear to complete—before, say, the CUDA runtime reports an error such as `CUDA_ERROR_ILLEGAL_ADDRESS` [22]. Some vulnerabilities are even *masked* [38], yielding zeros or silent data corruption, unless tools such as NVIDIA's Compute Sanitizer [21], cuCatch [19], or Triton-Sanitizer [39] (for kernels written in the Triton language) check memory accesses with additional instrumentation or analysis. Thus, the asynchronous nature of GPU memory faults, and the partial masking, complicate the root-cause analysis of memory errors in GPU code and motivate the need for GPU-aware fuzzers.

III. PRIOR WORK AND LIMITATIONS

Fuzz testing DL frameworks has been studied extensively, primarily targeting two categories of bugs: (a) *correctness bugs* (numerical/semantic inconsistencies) and (b) *reliability bugs* (crashes/memory errors). Correctness bugs include numerical and semantic inconsistencies between implementations of allegedly equivalent operations, or between implementations and their corresponding specifications. Reliability bugs include memory safety violations and crashes that can compromise system stability. Table I summarizes the detection capabilities of 22 state-of-the-art DL framework fuzzers, along with their oracles for the aforementioned bug categories, and the total number of bugs reported by each tool on TensorFlow. Overall, the six tools in the bottom group of Table I report no GPU or CPU runtime errors on TensorFlow. Four of them (EAGLE [55], MoCo [56], LEMON [57], and Muffin [58]) target correctness bugs and generate valid inputs, which rarely trigger runtime errors. On the other hand, tools with crash-oriented oracles report some runtime errors, but only a subset of them report runtime errors related to CPU memory safety (column `SIGSEGV`), and an even smaller subset related to GPU memory safety. Namely, ACETest [40] reports six GPU-specific errors and TitanFuzz [48] reports one.

Correctness-oriented Fuzzers. Tools such as FreeFuzz [50], EAGLE [55], Muffin [58], and NablaFuzz [51] focus on detecting output inconsistencies across backends or execution modes. To do so, they generate valid, shape-consistent inputs that satisfy API constraints. DeepREL [52] and MoCo [56] leverage API similarities and code reassembling for differential testing, while LEMON [57] mutates entire models. Since these tools prioritize *input validity* to enable differential oracles, they rarely exercise the invalid, boundary-case inputs needed to trigger memory safety violations, hindering their ability to detect GPU runtime errors. NeuRI [53], using inductive synthesis, captures some invalid traces and found one crash, but remains focused on correctness.

Crash-focused Fuzzers. These tools prioritize finding crashes but vary in effectiveness. ACETest [40] extracts constraints from kernel implementations to generate boundary-straddling inputs, detecting 137 TensorFlow bugs, including 6 GPU errors. IvySyn [23] and Orion [43] target low-level kernels or historical crash patterns, finding numerous CPU bugs (86 and 29, respectively) but struggling with GPU memory safety. Other approaches like DocTer [42], ConFL [54], and DeepConstr [49] utilize constraint extraction (from documentation or code) to generate inputs that are mostly valid. While they find CPU crashes, they lack the *specialized oracles* and *GPU state awareness* required to detect GPU memory corruption. Similarly, tools like SkipFuzz [45] (active learning), PathFinder [47] (concolic testing), and TensorScope [44] (cross-framework testing) optimize for coverage or API consistency while aiming to exercise deep kernel logic, lacking GPU memory error detection capabilities.

LLM-based Fuzzers. Recent tools like TitanFuzz [48], FuzzGPT [59], DFUZZ [46], and TransFuzz [61] leverage LLMs to generate high-quality seeds or unusual programs. While TitanFuzz found one GPU error, these approaches generally focus on producing valid, high-coverage inputs. They largely fail to uncover memory errors, as shown in Table I, due to their focus on expanding coverage and generating valid seeds (combined with the overhead of LLM-generated mutations).

The aforementioned fundamental gaps motivate the need for specialized GPU-oriented fuzzing approaches that: (i) can operate at the DL-kernel execution level with deep GPU context awareness; (ii) understand and model GPU-specific memory hierarchies and access patterns; and (iii) detect memory corruptions beyond observable crashes.

A concurrent work, GPU-Fuzz [62], also targets GPU memory-error discovery in DL frameworks. GPU-Fuzz employs operator-level constraint modeling and solver-guided test generation to explore parameter configurations that trigger low-level, CUDA-kernel faults. This approach is conceptually related to prior constraint-guided testing techniques, such as ACETest [40], but differs in the following key aspect: whereas ACETest automatically extracts input-validation constraints

from operator code, GPU-Fuzz relies on manually specified operator constraints. In fact, GPU-Fuzz reports 45 constraints across 13 operators and notes that extending coverage requires additional manual effort. In contrast, `FuzzWarp` adopts an automated, kernel-centric design that directly instruments native GPU kernels in DL frameworks (e.g., TensorFlow and PyTorch) and fuzzes them at the kernel boundary. This design eliminates manual operator modeling, scales to substantially broader low-level kernel coverage, verifies actual GPU dispatch at runtime, and improves the precision of attributing deferred GPU memory faults.

IV. Threat Model

We consider an adversary who seeks to exploit *GPU-implemented* kernels within DL frameworks (e.g., TensorFlow, PyTorch) by triggering memory-safety vulnerabilities in CUDA/PTX code. The adversary can supply *arbitrary* model inputs—images, tensors, serialized computation graphs, or launch-parameter blobs—that propagate from a high-level "wrapper" service to native kernels written in C/C++ with CUDA extensions, where language-level checks for spatial [63] and/or temporal [64] memory safety may be absent [14]. The adversary's objectives include corrupting GPU-resident data or control structures, overwriting device-to-host pointers, or forcing the CUDA runtime into error states, such as `CUDA_ERROR_ILLEGAL_ADDRESS`, thereby enabling arbitrary code execution, data exfiltration, and/or privilege escalation in multi-tenant environments [12], [13], [34]. Cloud inference platforms (e.g., NVIDIA Triton Inference Server [26], IBM Watson Discovery [65]) typify the victim scenario: a malformed request accepted by a wrapper microservice can propagate (often unchanged) to vulnerable GPU-kernel code [14].

By uncovering out-of-bounds, misaligned, and/or temporal memory errors in GPU-implemented kernel code, in an *automated* and *systematic* manner, `FuzzWarp` provides concrete, minimal proof-of-vulnerability inputs (§VI-D) that enable timely patches in framework codebases and, transitively, eliminate a class of exploits across applications built atop DL frameworks. Our adversarial capabilities are on a par with prior work in the field [12]–[14], [23], [34].

V. Fuzzing GPU Kernels in DL Frameworks

Earlier studies have extensively investigated how to effectively (and efficiently) uncover CPU-side memory errors in DL frameworks (§III), but their ability to uncover GPU-side errors is limited (see Table I). From our systematic analysis of 22 state-of-the-art DL-framework fuzzers, we have identified three main *challenges* when fuzzing GPU kernel implementations, which `FuzzWarp` aims to address.

Challenge #1: Ensuring GPU-kernel Execution. Modern DL frameworks are multi-backend: the same high-level operation may have multiple kernel implementations (CPU, GPU, *etc.*), and the framework *dynamically* selects which implementation to execute at runtime. Fuzzers operating, say, at the Python API level could repeatedly fail to exercise GPU kernels, diverting fuzzing effort away from GPU execution paths (unless the fuzzer explicitly verifies where execution actually occurs). As described in Section II-A, execution may occur either on the CPU or GPU depending not only on user directives (e.g., `tf.device`), but also on kernel availability and compatibility constraints. Crucially, mutations can *shift* an operation across backend boundaries: some mutated inputs may satisfy the GPU kernel's preconditions and result in executing the GPU variant, whereas others may violate `dtype`/shape/layout/attribute constraints and be routed to a CPU implementation (fallback) or an early runtime error.

Consequently, a GPU-centric fuzzer should be *placement-aware*: it needs to observe the framework's actual dispatch decision (which kernel variant was invoked) and steer fuzzing effort toward executions that truly reach GPU implementations, rather than assuming that high-level device directives imply true GPU kernel coverage. In Section IX-B, we quantify how often existing fuzzers miss their intended GPU kernel.

Challenge #2: Preserving Memory Placement. Beyond steering execution toward the GPU (challenge #1), a fuzzer must contend with the allocator-dependent constraints of the kernel's input. A kernel-level fuzzer must preserve this *allocator/placement fidelity* when creating mutated inputs: since mutation occurs after the framework completes device placement and transfers, the runtime does not repeat these steps for the mutated tensors. Allocating a tensor in the wrong memory domain can therefore cause host code to access device memory, GPU code to access host memory, or execution to fail before reaching the target GPU kernel. The first two cases can produce memory errors introduced by the fuzzing harness, while the third prevents the mutation from testing the target implementation. This requirement arises because DL frameworks allocate tensors using multiple memory domains (§II-C) and enforce per-argument placement constraints (e.g., `HOST_MEMORY`).

This issue is less visible for fuzzers operating through higher-level (e.g., Python) APIs, where the runtime performs placement and transfers before dispatch. In contrast, fuzzers that mutate inputs *after dispatch* must allocate mutated tensors via the framework's native allocators and preserve per-argument placement metadata; otherwise they risk false positives (invalid-device accesses), confounding both bug discovery and root-cause attribution.

Challenge #3: Pinpointing Offending Mutations. Unlike CPU memory errors, which often surface as *synchronous*, process-terminating signals (e.g., `SIGSEGV`, `SIGABRT`) at the faulting instruction, GPU memory errors may not immediately propagate to the host process. CUDA kernel launches are typically *asynchronous*: the host thread queues a kernel and continues execution while the GPU runs in the background. As a result, an out-of-bounds access inside a CUDA kernel may only be reported *later*, when the host reaches an explicit synchronization point (e.g., `cudaDeviceSynchronize`) or another CUDA API call that requires completion of prior work, at which point the runtime may return an error such as `cudaErrorIllegalAddress`.

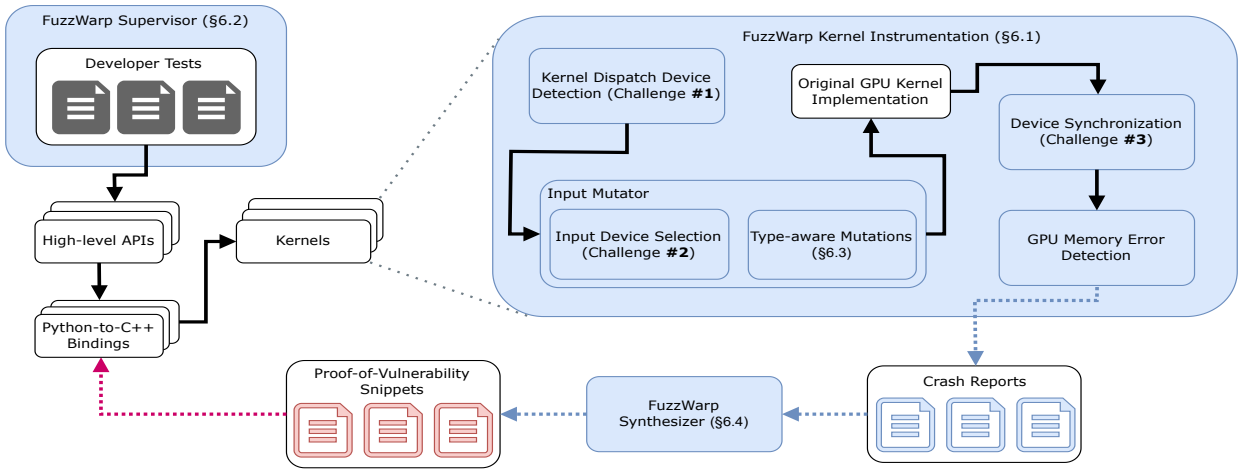


Fig. 1. Overview of FuzzWarp.

This *deferred error propagation* breaks the standard fuzzing assumption that a failing input causes an immediate, easily attributable runtime error: without synchronizing after each mutation, an error triggered by mutation M_i might be observed only after mutation M_{i+1} (or later), leading to misattribution and missed bugs. Consequently, correctly identifying offending mutations requires GPU-specific error-detection oracles with per-mutation synchronization barriers and checks for CUDA error codes, or the use of memory error sanitizers, which implicitly synchronize execution and provide enhanced memory error detection, albeit at extra performance overhead.

VI. FuzzWarp Design

FuzzWarp is a GPU-centric, mutation-based fuzzer designed to uncover memory-safety vulnerabilities in the GPU kernels of DL frameworks. Unlike conventional fuzzers that operate on high-level APIs [40], [43], [44], [48], FuzzWarp adopts a *bottom-up* design. First, it fuzzes and uncovers memory errors in the low-level GPU kernel implementations, and it then synthesizes Python snippets that trigger the uncovered memory errors from high-level, publicly-accessible APIs. As previous work has demonstrated [23], static type information available in low-level languages (e.g., C++), which are used for implementing kernels, provides opportunities to perform *type-aware mutations* based on tensor input types. Furthermore, directly operating on kernel code provides better control over which kernel implementations to fuzz, allowing FuzzWarp to focus only on GPU implementations. While FuzzWarp can be applied generally to different DL frameworks, we focus our design exposition on TensorFlow. In Section VII, we describe our adaptation of FuzzWarp to PyTorch.

Figure 1 provides an overview of the architecture of FuzzWarp. First, FuzzWarp instruments GPU-implemented kernels with *fuzzing wrappers* (§VI-A). Next, it uses the framework's existing developer tests as an execution harness, and seed corpus, to drive execution down to the instrumented kernels (§VI-B). When execution reaches the native code, FuzzWarp's instrumentation ensures that a GPU kernel implementation is dispatched, and then it runs a per-kernel mutation-based fuzzing loop that repeatedly mutates the original inputs (seeds) and re-invokes the kernel by generating {device, type}-aware mutations (§VI-C). To isolate kernels and preserve clean mutation attribution, FuzzWarp fuzzes one kernel at a time per process invocation. After each mutation, FuzzWarp enforces GPU synchronization and checks for CUDA errors to detect memory faults. Additionally, the instrumented framework runs under NVIDIA's Compute Sanitizer [21] to capture silent memory errors that may not surface through CUDA's runtime error reporting. Offending inputs that trigger GPU memory errors are logged in crash reports, which FuzzWarp's synthesizer (§VI-D) turns into *Proof-of-Vulnerability* (PoV) [23] snippets that trigger the underlying GPU memory error using public Python APIs.

Although FuzzWarp's mutation strategies, GPU synchronization, memory-error detection, and PoV validation do not depend on a particular DL framework, integrating FuzzWarp requires access to framework-specific interfaces. In particular, each implementation must identify the native operation entry points, intercept dispatch to determine whether execution reaches the GPU, allocate mutated tensors through the framework's native allocator, and map the intercepted operation back to a Python binding. We instantiate these interfaces for TensorFlow in what follows, and for PyTorch in Section VII.

A. GPU Kernel-code Instrumentation

To fuzz test native kernel code, FuzzWarp first identifies the functions that implement DL-framework kernels and then injects lightweight wrappers around them to enable input mutation and error detection during execution.

1) Kernel Extraction: In TensorFlow, each operation (§II-A) is executed by a kernel that inherits from either the OpKernel base class, for synchronous execution, or its asynchronous counterpart, AsyncOpKernel, implementing the Compute or ComputeAsync method, respectively.

These methods are invoked when the kernel is executed and encapsulate the kernel's core logic. Their first argument is a pointer to an `OpKernelContext` object, which provides metadata about the execution environment, like the target device and the allocator's attributes for the input/output tensors.

`FuzzWarp` uses static program analyses (on the respective DL framework source code) to enumerate *all* kernel implementations. It identifies classes that inherit from `(Async)OpKernel` and override the `Compute(Async)` method. Kernels whose implementations interact with shared state (e.g., via synchronization locks or `ResourceMgr` objects to manage resource lifecycle) or contain empty `Compute(Async)` methods are excluded from fuzzing, as they result in non-deterministic or superficial executions. Note that kernels that merely receive resource-typed inputs (as opposed to managing them) are still fuzzed: `FuzzWarp` resolves resource handles to their underlying tensor representations and applies mutations accordingly (§VIII).

2) Kernel Wrapping: After identifying the set of kernels to be fuzzed, `FuzzWarp` instruments each kernel's source code, injecting its fuzzing wrapper. Specifically, `FuzzWarp` first renames the `Compute(Async)` method of each kernel, which contains the kernel's implementation, to `do_<kernel_name>` (e.g., `GatherOp::Compute` is renamed to `GatherOp::do_GatherOp`). Next, `FuzzWarp` replaces the original body of the `Compute(Async)` method with its fuzzing wrapper, which contains code that is responsible for invoking its various components that are needed to ensure GPU kernels are properly fuzzed and memory errors are detected. An example of a fuzzing wrapper can be seen in Listing 2. In the injected wrapper, `FuzzWarp` first conducts various checks to ensure that fuzzing is skipped (by calling the original kernel implementation) if the kernel has already been fuzzed or to avoid recursive fuzzing (i.e., only fuzz a single kernel at a time), as seen in ln. 6–8 of Listing 2.

Device Detection. Since `FuzzWarp` is designed to be a GPU-centric fuzzer, we want to ensure that we only target GPU kernel implementations for fuzzing and skip CPU implementations, which have been extensively covered by previous work [23], [40], [43], [44], [48] (challenge #1). To address this challenge, `FuzzWarp` does *not* rely on high-level device directives to infer GPU execution. Instead, it confirms the framework's *actual* runtime dispatch decision at the point where kernel code is entered, after placement and kernel selection have already occurred. Concretely, `FuzzWarp` reads the operation's placement device from the `OpKernelContext` passed to `Compute(Async)` via `c->device()->device_type()` (ln. 12–13 of Listing 2). Since TensorFlow may select a `DEVICE_DEFAULT` fallback even on a GPU device, the fuzzer checks the matching registration using `FindKernelDef` (ln. 15–20 in the simplified listing; implemented in the fuzzer constructor). It fuzzes only if the context device is GPU and the registration specifies `DEVICE_GPU`; otherwise, it invokes the original implementation with the original arguments.

```
1  void Compute(OpKernelContext* c) override {
2
3    /* Skip fuzzing if we are already in a
4       fuzzing session or the kernel has
5       already been fuzzed */
6    if (fuzzwarp::already_fuzzing ||
7        fuzzwarp::was_fuzzed("GatherOp"))
8        goto skip;
9
10   /* Skip fuzzing if operation is not
11      placed on a GPU */
12    if (c->device()->device_type() != "GPU")
13       goto skip;
14
15   const KernelDef* k = nullptr;
16   auto s = FindKernelDef(DeviceType(DEVICE_GPU),
17       c->op_kernel().def(), &k, nullptr);
18   if (!s.ok() || !k ||
19       k->device_type() != DEVICE_GPU)
20       goto skip;
21
22   fuzzwarp::already_fuzzing = true;
23
24   /* Call fuzzer to create mutation pools */
25   fuzzwarp::Fuzzer fuzzer =
26       fuzzwarp::Fuzzer("GatherOp", c);
27   tensorflow::OpKernelContext *fuzz_ctx;
28
29   /* Iterate through the mutation pools,
30    * invoking the original kernel each time */
31   while (fuzzer.has_more_mutations(true)) {
32     fuzz_ctx = fuzzer.get_fuzzed_context();
33     do_GatherOp(fuzz_ctx);
34     /* Synchronize and check
35      * for GPU memory errors */
36     cudaError_t err = cudaDeviceSynchronize();
37     if (err == cudaErrorIllegalAddress
38       || /* Other error checks */ ) {
39       fuzzer.log_gpu_error();
40       exit(-1);
41     }
42   }
43   fuzzwarp::already_fuzzing = false;
44 skip:
45   /* Return the original kernel result */
46   do_GatherOp(c);
47 }
```

Listing 2: Simplified `FuzzWarp`-instrumented version of the `GatherOp` kernel (`Compute` method).

Input Mutator. If `FuzzWarp` determines that the kernel's execution is dispatched to the GPU, it passes the original kernel inputs to the input mutator (ln. 25–26), which is responsible for: (1) identifying which device each input should reside on; and (2) generating mutation pools for each input on the appropriate device (and memory type). Specifically, the input mutator iterates through each input and first checks the device it resides on using the input's allocator attributes. It then checks the data type of each input and generates a mutation pool, which is allocated with the appropriate memory allocator, ensuring tensor allocation on the correct device (challenge #2).

Device Synchronization. Because GPU operations execute asynchronously, memory errors may not surface immediately after occurring. To ensure deterministic error detection and correct mutation attribution (challenge #3), `FuzzWarp` enforces a per-mutation GPU error oracle.

Before invoking the kernel on a mutated input, the wrapper clears any pending CUDA error state (e.g., by querying the last error), then launches the kernel and inserts a synchronization barrier by invoking `cudaDeviceSynchronize` (ln. 36). This call blocks the host thread until all previously issued work in the active device stream has completed, forcing deferred GPU faults to be reported *before* the next mutation begins. `cudaDeviceSynchronize` returns a `cudaError_t` status code, which `FuzzWarp` inspects to identify GPU memory faults (e.g., `cudaErrorIllegalAddress`, ln. 37). When such an error is observed, `FuzzWarp` records the current mutation as offending input (rather than incorrectly blaming a later mutation), creates a marker file indicating that a GPU-side memory error occurred (ln. 39), and errors out the process (ln. 40) to trigger crash handling by the supervisor.

Silent Error Detection. While device synchronization reliably captures faults that propagate through CUDA's runtime error reporting, some GPU memory errors may manifest as silent corruption without an immediate `cudaError_t`. For these cases, `FuzzWarp` additionally employs instrumentation-based tools (e.g., Compute Sanitizer [21]). `FuzzWarp` checks only for memory-safety errors; detecting data races or missing synchronization requires GPU race detectors [66], [67].

Kernel Synchronization. `FuzzWarp` injects additional code in asynchronous kernels (i.e., those implementing the `ComputeAsync` method) to ensure that each mutation executes synchronously. Although asynchronous execution occurs in both CPU and GPU kernels, deterministic fuzzing and reliable error detection require strictly ordered execution. `FuzzWarp` modifies `ComputeAsync` to behave synchronously by injecting code that executes each mutation in a dedicated thread. The main fuzzing thread then waits for the completion of each mutation before proceeding to the next, guaranteeing deterministic execution. This synchronization is confined to the instrumented fuzzing wrapper and does not alter kernel computation, inputs, or outputs. It only constrains host-side scheduling to attribute faults to the correct mutation.

B. Force-executed GPU Kernel-code Fuzzing

DL framework kernels are not publicly accessible, but are instead invoked through higher-level APIs (e.g., Python APIs). Since our fuzzing wrappers are injected around native kernel code, `FuzzWarp` needs to cause execution to reach such kernels with a proper calling context. Some kernels are more challenging to invoke, as they call the GPU implementation only for specific input arguments with particular data types (e.g., floating-point tensors). Fortunately, most frameworks (including TensorFlow) provide comprehensive test suites that cover a wide range of kernel functionalities, which `FuzzWarp` leverages to force-execute the instrumented kernels. Specifically, a supervisor process enumerates all publicly available Python tests provided by TensorFlow and executes them, monitoring for crashes or abnormal terminations that indicate a memory error. When a GPU kernel is invoked during testing, `FuzzWarp`'s instrumentation intercepts the call and triggers the fuzzing process as described in Section VI-A.

During fuzzing, `FuzzWarp` assigns a unique identifier to each mutated input, which it logs alongside the fuzzed kernel name. The supervisor continuously monitors the exit codes of the executed tests and, when it observes an exit code indicating a crash (§VI-A2), it re-executes the crashing test. Upon re-execution, the kernel that triggered the memory error is invoked again, where `FuzzWarp`'s fuzzer retrieves the last mutation executed (i.e., the one that caused the memory error) and records the offending arguments and kernel name to a *crash report*. Error-indicating exit codes do not necessarily occur only when `FuzzWarp` exits after detecting a GPU memory error—e.g., they could be due to software-caused aborts or floating-point exceptions. Since `FuzzWarp` specifically targets GPU memory errors, it terminates the fuzzing campaign for the crashing kernel only when a GPU memory error was detected (indicated by a file created by the fuzzer), or when a pre-determined (configurable) number of consecutive crashes occur in the same kernel without a GPU memory error.

C. Mutation Strategies

`FuzzWarp` uses mutation-based fuzzing to identify memory errors in GPU kernels. It leverages the type information available in the statically-typed C++ kernel code, where it injects its fuzzing wrappers (§VI-A), to construct *type-aware* mutations based on the input tensors' data types and shapes.

Through empirical analysis of Common Vulnerabilities and Exposures (CVEs) and bug reports specific to DL-framework code, along with insights from CUDA kernel implementations, we have identified several mutation strategies that reliably trigger memory errors in GPU kernels. Our studies reveal that memory errors often result from edge-case inputs, such as tensors containing extremely large dimensions, tensors with zero-sized dimensions, and large negative/positive values. `FuzzWarp` currently supports the following mutation types:

Dimension-product Overflow Mutations. This strategy systematically explores integer overflow vulnerabilities arising from dynamic tensor shape calculations. Given an input tensor with shape $\langle d_1, d_2, ..., d_n \rangle$, the fuzzer computes scaled dimensions $\langle d'_1, d'_2, ..., d'_n \rangle$ such that $\prod_{i=1}^{n} d'_i \approx 2^{31}$ or $\prod_{i=1}^{n} d'_i \approx 2^{31} - 1$ (`INT32_MAX`). `FuzzWarp` does not mutate every dimension independently. For each mutation, it scales the dimensions so that the resulting product reaches the target overflow range. This preserves the rank and structural properties of the original tensor while targeting a critical vulnerability class: operations that iteratively compute total element counts via dimension multiplication can experience silent integer overflows during intermediate products, even when individual dimensions pass validation checks. This approach specifically targets kernel implementations that unsafely cast 64-bit shape calculations to 32-bit integers for indexing or looping.

Since these mutations trigger large allocations (e.g., up to 16GB per tensor) that may lead to GPU memory exhaustion, `FuzzWarp` performs *type-aware memory feasibility checks*: before instantiating each mutated tensor, it estimates the total memory footprint as $\text{element_count} \times \text{element_size_bytes}$, where the per-element cost depends on the tensor's *data type*

```
1  // maps a warp to each row
2  template <typename T, typename OUT_T, typename Op>
3  __global__ void __launch_bounds__(1024)
4  RowRedKernel(T in, OUT_T out,
5    int num_rows, int num_cols, Op op,
6    typename
7      iterator_traits<T>::value_type initVal) {
8
9    ...
10   assert(blockDim.x % WARPSIZE == 0);
11   int warps_per_blck = blockDim.x / WARPSIZE;
12   int warp_index = threadIdx.x / WARPSIZE;
13   int row = blockIdx.x *
14     warps_per_blck + warp_index;
15   int lane = threadIdx.x % WARPSIZE;
16
17   if (num_cols == 1) {
18     int gid = threadIdx.x +
19         blockIdx.x * blockDim.x;
20     if (gid < num_rows) out[gid] = in[gid];
21     return;
22   }
23 }
```

Listing 3: TensorFlow CUDA kernel (`RowRedKernel`) vulnerable to an integer overflow.

```
1  template <typename dtype>
2  __global__ void MaxPoolGradBackward(
3      const int nthreads,
4      const dtype* top_diff,
5      const int64* mask,
6      const int top_offset,
7      const int bottom_offset,
8      dtype* bottom_diff,
9      const bool include_batch_in_index) {
10   GPU_1D_KERNEL_LOOP(index, nthreads) {
11     const int offset = include_batch_in_index
12         ? 0
13         : (index / bottom_offset) * top_offset;
14     bottom_diff[index] =
15         top_diff[offset + mask[index]];
16   }
17 }
```

Listing 4: TensorFlow CUDA kernel (`MaxPoolGradBackward`) vulnerable to an out-of-bounds access.

```
1  T=DT_BOOL,
2  _device='/job:localhost/replica:0/task:0/device:GPU:0'
3  Tensor<type: bool shape: [2147483647] values: 1 1 1...>
```

Listing 5: Crash report for `WhereGPUOp` kernel.

(e.g., 4 bytes for `float32`, 8 bytes for `float64`). This computed footprint is compared against the device's available GPU memory, and mutations are only applied if the tensor fits within device constraints.

Listing 3 illustrates an example of a kernel where `FuzzWarp` has uncovered a GPU memory error using this mutation strategy. Specifically, in ln. 18–19, the `gid` variable is declared as `int32` and computed as `gid = blockIdx.x * blockDim.x + threadIdx.x`. Here, `blockIdx.x` and `blockDim.x` are CUDA built-in variables determined by the kernel's launch configuration, which can be influenced by the dimensions of the input tensor. The value of `gid` is then used to index into the input tensor (ln. 20). When the input tensor has a very large dimension, the product `blockIdx.x * blockDim.x` may overflow the 32-bit integer range and result in a negative `gid` value. Using such a negative index causes an out-of-bounds access.

Boundary-value Mutations. This generates tensors populated entirely with extreme and edge-case values tailored to each data type. For integer types, values include type limits (`INT32_MIN`, `INT32_MAX`, `LLONG_MIN`, `LLONG_MAX`), negative indices (-1), and boundary-crossing values (e.g., `INT32_MAX`$+1000$ as `int64`). For floating-point types, the mutation injects `NaN`, zero, small magnitudes (e.g., $2.50933511 \times 10^{-20}$), and large magnitudes (e.g., $1.15215043 \times 10^{25}$, $8.11656739 \times 10^{20}$). Complex types receive combinations of these edge cases in real and imaginary components. String types are populated with empty strings, negative numeric strings, and invalid representations. These values are applied uniformly across all tensor elements while preserving the original shape structure. This mutation is particularly effective against operations that use tensor values as indices, sizes, or loop bounds, without proper range validation, as well as kernels performing pointer arithmetic that assumes well-behaved numerical inputs.

For example, in Listing 4, the contents of the `mask` tensor (cast to an `int64` pointer in the code) are used to index the `top_diff` tensor (ln. 14–15) without validation. Boundary-value mutations that populate `mask` with negative values (e.g., -1) or values exceeding the `top_diff` array bounds trigger out-of-bounds GPU memory access.

Zero-dimension Mutations. This mutation strategy generates tensors with at least one dimension set to zero. The mutation creates both fixed-rank patterns (e.g., $[0]$, $[B, 0]$, $[B, H, W, 0]$) and rank-preserving variants, where a zero is inserted at strategic positions (first, last, or middle dimensions) while maintaining the original tensor's rank. For example, given a seed tensor with shape $[B, H, W, C] = [32, 224, 224, 3]$, `FuzzWarp` generates rank-preserving zero-dimension variants such as $[32, 0, 224, 3]$, $[32, 224, 0, 3]$, and $[32, 224, 224, 0]$.

When initializing the mutation pools for a target kernel, `FuzzWarp` inspects the native-level input argument types and constructs a corresponding pool of mutation values based on the strategies described above. These values are stored in per-device mutation pools, ensuring that all generated mutations are allocated on the appropriate device. This device-aware allocation is essential to maintain compatibility with the target kernel's execution context and to prevent false positives caused by cross-device mismatches. Once the mutation pools are initialized, the fuzzer lazily enumerates all feasible value combinations and applies them until either a user-defined timeout is reached or all combinations have been exhausted.

D. Synthesis of Vulnerability-triggering Code

Once `FuzzWarp` identifies all the memory errors through fuzzing, it synthesizes Proof-of-Vulnerability (PoV) [23] snippets that reproduce each crash using public TensorFlow Python APIs. This synthesis bridges the gap between low-level kernel inputs and user-facing code, enabling developers to verify and fix the underlying issues.

```
1  import tensorflow as tf
2
3  with tf.device('/GPU:0'):
4    condition = tf.constant(True,
5      shape=[2147483647], dtype=tf.bool)
6    tf.raw_ops.Where(input=condition)
```

Listing 6: FuzzWarp PoV for the WhereGPUOp kernel.

The synthesis process begins with the crash report logged during fuzzing, which contains the attributes and mutated tensor inputs that triggered the error (Listing 5). During kernel registration, FuzzWarp records the mapping between each C++ kernel class and the TensorFlow operation(s) it implements. For each candidate operation, the synthesizer resolves its public tf.raw_ops binding and inspects the binding's signature to obtain the expected argument names and order. It matches logged operation attributes to the corresponding named parameters and assigns the remaining tensor inputs in the order recorded by the crash report.

The synthesizer converts internal data types (e.g., DT_BOOL) to their Python equivalents (e.g., tf.bool) and reconstructs each tensor from its recorded data type, shape, and value representation. It uses tf.constant (ln. 4–5), or tf.zeros/tf.ones for uniform tensors. Resource inputs are reconstructed using tf.Variable and passed to the operation through their handles. In Listing 6, this process maps WhereGPUOp to tf.raw_ops.Where (ln. 6) and reconstructs its boolean input.

To request GPU dispatch, the synthesizer wraps the operation in a device context manager (e.g., with tf.device("/GPU:0")). Inputs declared as host memory remain handled by TensorFlow's placement. When a kernel implements multiple operations, the synthesizer generates a candidate PoV for each public binding. Finally, we execute the synthesized PoVs using the upstream TensorFlow version combined with NVIDIA's Compute Sanitizer [21] and retain the PoVs that reproduce the GPU memory error in a standard Python environment, which also confirms that the call reached the GPU kernel.

VII. Framework Generalization

To examine whether FuzzWarp can be applied beyond TensorFlow, we implemented FuzzWarp for PyTorch v2.13. As discussed in Section VI, the mutation strategies, GPU synchronization, memory-error detection, and PoV validation are framework-agnostic; only the interfaces that enumerate native operations, intercept dispatch, allocate tensors, and resolve Python bindings must be adapted.

PyTorch implements tensor operations through ATen, its C++ tensor library, which defines operator schemas and dispatches each operator to a device-specific implementation. Instead of instrumenting (Async)OpKernel methods, the PyTorch implementation instruments the generated ATen call and redispatch wrappers, and invokes the fuzzing loop when PyTorch's dispatcher selects CUDA as the respective execution backend.

Because the hook is inserted in the generated C++ wrapper, FuzzWarp receives each argument in its declared C++ type and constructs mutated tensors through ATen's allocation interfaces—the counterpart of TensorFlow's allocator-attribute path in Section VI-C. FuzzWarp synchronizes GPU execution after every invocation. For PoV synthesis, it uses ATen schemas to reconstruct the failing Python calls and validates them on stock PyTorch under Compute Sanitizer.

VIII. FuzzWarp Implementation

We implemented FuzzWarp in ≈5.4K lines of C/C++ code and ≈850 lines of Python code for TensorFlow, and in ≈9.5K lines of C/C++/CUDA code and ≈5.3K lines of Python code for PyTorch (§VII). More specifically, we built FuzzWarp on top of the IvySyn framework [23]. Note that IvySyn is a generic DL-framework fuzzer, which, however, lacks support for fuzz testing GPU-implemented kernels due to the issues outlined in Section V. The PyTorch implementation adapts FuzzWarp's mutation strategies and fault-attribution approach to PyTorch, using generated dispatcher hooks, native tensor allocation, and schema-driven PoV synthesis.

GPU Memory-error Detection. We built GPU-specific components to address the asynchronous nature of GPU memory errors. First, we implemented dynamic kernel-code dispatch device detection by inspecting the OpKernelContext and resolving the matching kernel registration with FindKernelDef; we fuzz only when the context device is GPU and the registration specifies DEVICE_GPU. Second, we implemented input device detection logic that examines the allocator attributes of each input tensor to determine its memory placement and ensure mutations are allocated on the correct device. Third, we implemented GPU synchronization primitives by injecting cudaDeviceSynchronize() calls after each mutation and inspecting the returned cudaError_t status codes to detect memory violations (e.g., cudaErrorIllegalAddress and cudaErrorMisalignedAddress). Lastly, we added NVIDIA's Compute Sanitizer [21] to the fuzzing workflow to capture detailed GPU memory-error reports during PoV validation; we also integrated it into the fuzzing pipeline to capture silent GPU errors.

Device-aware Allocation. We implemented device-aware tensor allocation for FuzzWarp's mutation engine. The mutator maintains per-device mutation pools for each tensor data type, examining each input tensor's allocator attributes and generating mutations allocated with the appropriate memory allocator (i.e., CPU, pinned host, or GPU device memory).

Additional Mutations. We implemented a new GPU-focused mutation type: dimension-product overflow mutations (≈600 LOC) that scale tensor dimensions so that their cumulative product approaches INT32_MAX to trigger integer overflows in index calculations on GPU code implementations. The mutator performs shape-level scaling over the input tensor's dimensions (rather than mutating every dimension independently) while preserving its rank.

To avoid exceeding device memory limits, the mutator estimates the total memory footprint of the mutated tensor based on its data type size and only applies such mutations if the resulting tensor fits within the available device memory.

Extended Data-type Support. We implemented support for a broad range of TensorFlow data types including `Complex` type and `Resource` tensors, which IvySyn lacked. Resource tensors are handles to stateful objects managed by TensorFlow's Resource Manager (e.g., variables, tensor arrays, lookup tables). These objects encapsulate mutable state and may internally own tensor buffers, making resource tensors fundamentally different from standard tensors. To handle them correctly, we added specialized detection for resource-typed inputs that resolves their handles via TensorFlow's resource APIs, inspects the underlying object's data type, and mutates the associated tensor buffers accordingly.

Enhanced PoV Synthesis. We implemented a GPU-aware PoV synthesizer on top of IvySyn's synthesizer (≈600 LOC). `FuzzWarp`'s synthesizer wraps generated API calls in `tf.device("/GPU:0")` context managers to ensure GPU dispatch, while preserving the exact tensor shapes, data types, and values from crash reports, adding extra support for `Resource` and `Complex` types. We integrated automatic PoV validation by executing synthesized snippets under Compute Sanitizer, and verifying that the GPU memory error can be reproduced in TensorFlow v2.20.

Clang/LLVM Pass Extensions. We modified IvySyn's Clang-based source transformation pass (≈250 LOC) to support instrumentation of both synchronous (`Compute`) and asynchronous (`ComputeAsync`) TensorFlow kernels.

IX. Evaluation

We evaluate the effectiveness of `FuzzWarp` in uncovering GPU memory errors by comparing it with four state-of-the-art fuzzers for DL frameworks across multiple TensorFlow versions (§IX-B). Furthermore, we assess `FuzzWarp`'s capability to detect GPU memory errors in TensorFlow v2.20 and PyTorch v2.13, and evaluate PoV synthesis in both (§IX-C). Note that `FuzzWarp` is a white-box mutation-based fuzzer that is agnostic to coverage feedback; it instead emphasizes structured edge-case exploration.

A. Experimental Setup

Fuzzer Selection. We compared `FuzzWarp`'s ability to detect GPU memory errors against other DL framework fuzzers. In particular, from the set of state-of-the-art fuzzers shown in Table I, we selected those that (a) demonstrated some ability to uncover memory errors (either CPU-side memory errors that manifest as segmentation faults, or GPU-side memory errors that manifest as CUDA memory faults), and (b) have code publicly-available, allowing us to reliably benchmark them. Four fuzzers fulfill these criteria:

- ACETest [40]: a constraint-based operator fuzzer that derives input constraints from validation code to generate valid but bug-triggering tensors.
- TitanFuzz [48]: an LLM-guided fuzzer that synthesizes and mutates test programs under syntax and shape constraints to uncover runtime failures.
- Orion [43]: a history-driven fuzzer that learns from past bugs to guide input mutations.
- TensorScope [44]: a differential testing tool that generates calls to semantically equivalent APIs to detect inconsistencies across DL frameworks.

To ensure a fair comparison against all selected fuzzers, we back-ported our prototype implementation to the specific versions each tool was originally evaluated on (see "Version Alignment" below). Furthermore, in the context of each experiment, we selectively constrained `FuzzWarp` to a respective subset of common kernels (see "Kernel-subset Selection" below). We did not include comparisons against IvySyn [23], as adding support for GPU memory error detection necessitates re-engineering the tool into a device-aware framework, involving non-trivial changes beyond its original design (effectively retrofitting a weaker version of `FuzzWarp` to IvySyn). Similarly, we excluded a comparison against GPU-Fuzz [62], owing to both its limited operator coverage and its code not being publicly available.

Version Alignment. To ensure a fair comparison, we evaluated each fuzzer on the TensorFlow version its prototype was implemented for and we back-ported our `FuzzWarp` prototype to the same TensorFlow version for each comparison. Specifically, ACETest was evaluated on TensorFlow v2.9 (commit `8a20d54a`), TitanFuzz and TensorScope on v2.11 (commit `d5b57ca9`), and Orion on v2.13 (commit `1cb1a030`).

Kernel-subset Selection. The DL framework fuzzers we compared against test TensorFlow via high-level, Python APIs, while `FuzzWarp` directly fuzzes the low-level kernel implementations. Even though the invocation of a high-level API will eventually result in calling low-level kernel code, there is no clear one-to-one mapping between the two levels. Since the goal of our experiment is to investigate how capable each tool is at uncovering GPU memory errors in low-level GPU kernel implementations, we need to ensure that, for each comparison, `FuzzWarp` fuzzes the same set of kernels as the compared-against fuzzer. To do so, we modified `FuzzWarp`'s instrumentation to have it log each low-level kernel invocation. We then ran each fuzzer under its default configuration and collected the set of kernels that are (implicitly) invoked by the fuzzer. For a fair comparison, we then restricted `FuzzWarp` to fuzz only the same subset of kernels when comparing against each tool. In the case where a kernel is mapped to multiple high-level APIs, we randomly select one of the APIs.

Hardware Testbed. For the comparison of `FuzzWarp` against ACETest, TitanFuzz, Orion, and TensorScope, we used a pool of four machines armed with 11 NVIDIA RTX A4000 [68] GPUs, with each fuzzer running in an isolated Docker container pinned to a single GPU. For the `FuzzWarp` fuzzing campaign on TensorFlow v2.20, we used a single host armed with five NVIDIA A40 [69] GPUs—all campaigns containerized, each assigned one GPU. The OS used for these experiments was Ubuntu 22.04.

For PyTorch v2.13, we used five Docker containers, each pinned to a separate NVIDIA A40 GPU with 48GB of memory. We built PyTorch from commit `cf30153c` with CUDA 12.6 and enabled Compute Sanitizer during fuzzing.

Kernel Coverage. Out of 357 GPU kernel implementations registered by 710 GPU operations in TensorFlow v2.20, `FuzzWarp` instrumented 316 implementations. Of the remaining 41, most were outside the pass's scope or could not be parsed because required headers were missing; 3 used the shared resource manager; and 1 had an empty implementation. During our five campaigns, 267 implementations reached the fuzzer. Of the other 49, 8 exposed no input arguments, 13 required unsupported references or resources, 9 were intentionally excluded, 6 (NCCL) implementations were not included in the build, and 3 registrations were outdated. Of the remaining 10, 4 were covered only by CPU tests, 4 had GPU tests but did not reach the instrumented implementation, and 2 had tests that selected a different GPU implementation.

For PyTorch v2.13, `FuzzWarp` fuzzed 708 of 802 distinct base ATen operators with CUDA implementations (88%) across the five campaigns. Of the other 94: 14 cannot run on our setup (10 require AMD GPUs and 4 need FP8 support that our GPU lacks), 9 are tensor-creation and copy operators that we exclude because mutating them breaks our test setup, and 8 take quantized tensors, which `FuzzWarp` does not support; 17 are never called on CUDA in our setup; and of the remaining 46, 28 were missed because of two prototype limitations that we have since addressed—a bug in how `FuzzWarp` recorded the shape of undefined and nested tensor arguments, and the lack of instrumentation on PyTorch's Python operator interface—, 12 are exercised only by `test_sparse.py` and 4 only by `test_transformers.py`, both of which our driver terminated early, while the remaining 2 were not targeted when their respective test executed.

B. Comparison with Existing Fuzzers

We compare the effectiveness of `FuzzWarp` in uncovering GPU memory errors against four state-of-the-art fuzzers: ACETest, TitanFuzz, Orion, and TensorScope. Each tool was executed for five independent iterations under identical conditions to account for non-determinism in mutation strategies [70]. `FuzzWarp` was evaluated on the same TensorFlow version and kernel subset used by each baseline. Table III summarizes the common kernel sets used. The kernels not covered by `FuzzWarp` fall into three categories: (i) zero-argument kernels, (ii) kernels not exercised by TensorFlow's test drivers, and (iii) kernels explicitly excluded from our fuzzing scope due to compatibility issues (e.g., stateful ops). We allocated a 20-minute timeout per kernel, resulting in up to three days of total execution time per run, depending on kernel-set size. Note that we count one unique crash per TensorFlow kernel (crash deduplication)—if a kernel exhibits multiple memory errors because of different root causes or access patterns, we report only the first detected error.

Table II summarizes the number of GPU memory errors detected by each tool. Overall, `FuzzWarp` outperforms all other fuzzers across every comparison, recording 48 GPU memory-error detections across the four comparisons, compared to 3 by the baselines. `FuzzWarp` detected seventeen more unique memory errors than ACETest. TitanFuzz detected one GPU memory error, whereas `FuzzWarp` found sixteen in the same kernel set. Orion uncovered one unique GPU memory error, while `FuzzWarp` detected five in the same set. This smaller margin is primarily due to Orion's limited kernel coverage. TensorScope failed to detect any GPU memory error, while `FuzzWarp` found nine under identical conditions. Of the three GPU memory errors uncovered by the other fuzzers, `FuzzWarp` missed the one found by TitanFuzz, in `Multinomial`.

GPU-kernel Reachability. To quantify how often inputs generated by existing fuzzers reach their intended GPU kernel (challenge #1), we re-ran each baseline with `FuzzWarp`'s kernel-invocation logging enabled (§IX-A). This instrumentation records the kernel implementation and device selected by TensorFlow while leaving each baseline's input-generation strategy unchanged. We classify each mutation into one of five outcomes: execution of the intended GPU kernel (*target GPU*), execution of another GPU kernel or helper (*other GPU/helper*), execution of the CPU implementation (*CPU fallback*), failure before dispatch (*no dispatch*), or termination due to a timeout, crash, or out-of-memory error. Table IV reports the results. Between 28.8% (TensorScope) and 44.3% (Orion) of the generated mutations do not execute the intended GPU kernel. The main source of off-target execution differs across tools. For TensorScope and ACETest, 27.8% and 20.4% of mutations, respectively, execute another GPU kernel or helper, while an additional 18.0% of ACETest mutations fall back to the CPU. In contrast, 29.3% of TitanFuzz mutations and 30.1% of Orion mutations fail before dispatch. Overall, executions that do not reach the target GPU kernel account for 24.3–85.0% of the measured execution time. `FuzzWarp` avoids this loss by verifying the dispatch decision before starting the fuzzing loop and directly invoking the selected GPU implementation for each mutation.

Asynchronous Fault Attribution. To isolate the effect of `FuzzWarp`'s per-mutation synchronization and error-checking oracle (challenge #3) from the rest of our design, we retrofit it onto the baseline fuzzers and measure their bug-finding capability independently of their original mutation engines. Out-of-the-box ACETest reported an average of 8.6 flagged kernels per campaign, of which 88% were non-reproducible, consistent with attribution drift from deferred GPU faults. After applying our oracle, ACETest identifies 10 unique GPU memory errors across its five runs, compared to the one confirmed error reported in its original configuration on the same kernel set (Table II). Enhanced Orion surfaces one additional unique error. With the same oracle, TensorScope identifies three unique GPU memory errors. `FuzzWarp` found one of them in its matched TensorFlow v2.11 campaign; we discuss the other two below.

TABLE II
COMPARISON OF FUZZWARP WITH STATE-OF-THE-ART FUZZERS.

	(TF 2.9)		*(TF 2.11)*		*(TF 2.11)*		*(TF 2.13)*	
Run	ACETest	**FuzzWarp**	TitanFuzz	**FuzzWarp**	TensorScope	**FuzzWarp**	Orion	**FuzzWarp**
1	1	10	1	8	0	5	0	3
2	1	12	0	8	0	5	1	3
3	1	10	0	6	0	4	0	3
4	1	6	0	10	0	6	1	4
5	1	10	0	8	0	4	1	5
Errors (unique)	1	**18**	1	**16**	0	**9**	1	**5**

TABLE III
COMMON KERNEL SETS USED FOR THE COMPARISON OF FUZZWARP AGAINST EXISTING FUZZERS.

TF version	Baseline	Reach	Common
2.9	ACETest	215	**174**
2.11	TensorScope	146	**125**
2.11	TitanFuzz	240	**214**
2.13	Orion	105	**99**

TABLE IV
GPU KERNEL REACHABILITY VS. OFF-TARGET EXECUTIONS IN BASELINE FUZZERS.

Outcome	**Tensor Scope**	**Titan Fuzz**	**ACE Test**	**Orion**
Target GPU	71.17	61.84	61.04	55.68
Other GPU/helper	27.83	3.96	20.39	13.58
CPU fallback	0.95	4.83	18.04	0.67
No dispatch	0.04	29.32	0.53	30.06
Timeout/crash/OOM	0.00	0.04	<0.01	<0.01
Not target GPU (mut.)	**28.83**	**38.16**	**38.96**	**44.32**

These results indicate that per-mutation synchronization and CUDA error checking account for a substantial portion of the bug-finding gap between CPU- and GPU-oriented DL fuzzing, independently of FuzzWarp's mutation/instrumentation.

While FuzzWarp outperformed the baselines overall, it missed seven bugs that the oracle-enhanced fuzzers found in the matched campaigns: four from ACETest, one from Orion, and two from TensorScope. Our campaign logs show four reasons for these misses. First, high-level TensorFlow APIs sometimes accept additional arguments that are not passed as arguments in the OpKernelContext object of the kernel, but are rather static attributes of the OpKernel class, which are not fuzzed by FuzzWarp. For one kernel, the test that reached it set such an attribute so that the kernel validates its inputs, which prevented FuzzWarp from triggering the bug found by both ACETest and TensorScope. Second, some bugs are triggered only by inputs of a specific shape. FuzzWarp exhausted all generated mutations of two kernels that share the same GPU code without producing that shape, missing the two bugs found by ACETest; TitanFuzz found the same bug in Table II. Third, for the bug found by both ACETest and Orion, mutations with extreme values either crashed the process outside the GPU, until FuzzWarp exceeded its limit of non-GPU-related crashes, or ran until the timeout. Finally, some kernels reach the 20-minute allocated timeout before exhausting all generated mutations. For the second TensorScope bug, FuzzWarp generated the triggering value, but only combined it with values of another argument that failed before reaching the GPU code. In separate TensorFlow v2.20 campaigns, FuzzWarp found this bug and a different bug in the kernel of the first TensorScope bug.

C. GPU Memory Errors in TensorFlow and PyTorch

To evaluate FuzzWarp's ability to uncover GPU memory errors in TensorFlow, we fuzzed TensorFlow v2.20 (commit 72fbba3d) with five independent instances running in parallel, each with a different seed, and enforced a 20-minute timeout for each TensorFlow kernel. All five instances finished within 3.5 days. During these runs, FuzzWarp fuzzed a total of 267 unique GPU TensorFlow kernel implementations, *uncovering 30 new GPU memory errors in unique CUDA kernel implementations*, as shown in Table V. For each of the 30 unique GPU memory errors, FuzzWarp successfully synthesized a PoV, yielding a PoV-synthesis success rate of 100%. Since FuzzWarp reports an issue only when the synthesized PoV is reproducible, we observed no false positives in our evaluation. In general, PoV synthesis can fail when the synthesizer cannot map a low-level kernel name back to its corresponding Python API (or cannot infer required API inputs/attributes); these cases did not arise here. Figure 2 and Figure 3 show, for TensorFlow and PyTorch, the cumulative distribution of the time FuzzWarp needed to trigger each GPU memory error. We measured an average throughput of 12.3 kernel executions per second for TensorFlow, and 68.8 operator executions for PyTorch, over the time spent fuzzing kernels, inclusive of device synchronization and Compute Sanitizer overheads. Note that each execution applies one mutation and calls the kernel/operator once.

For PyTorch v2.13, we ran five independent campaigns in parallel, each with a different seed and a 20-minute timeout per operator; all five finished within 4.6 days. In total, FuzzWarp fuzzed 708 ATen operators and uncovered GPU memory errors in 69 unique CUDA kernel implementations (Table V). For each of the 69, FuzzWarp synthesized a PoV that reproduces the error on PyTorch v2.13.

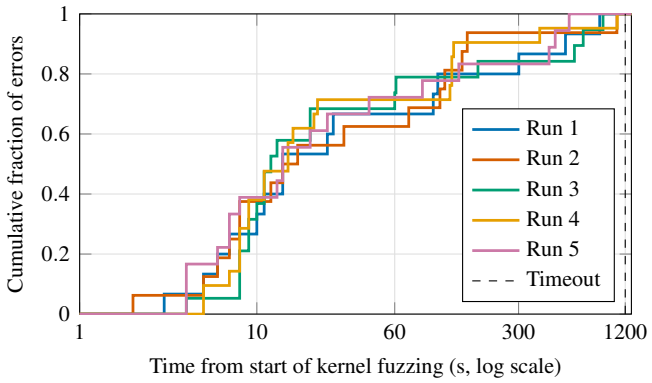


Fig. 2. CDF of the time `FuzzWarp` took to trigger each GPU memory error in our TensorFlow v2.20 campaign, measured from when it started fuzzing the affected kernel (20-minute timeout per kernel). Errors found in under 1 s are plotted at 1 s.

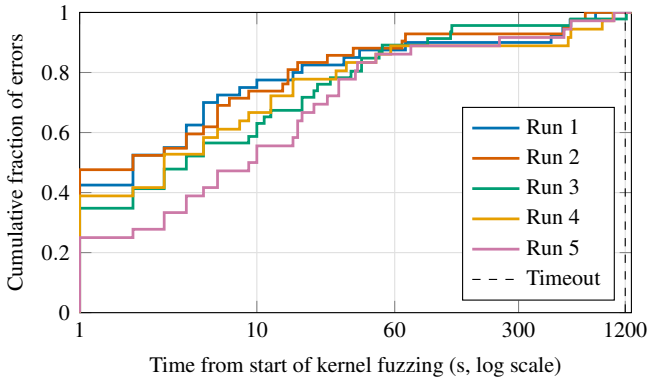


Fig. 3. CDF of the time `FuzzWarp` took to trigger each GPU memory error in our PyTorch v2.13 campaign, measured from when it started fuzzing the affected operator (20-minute timeout per operator). Errors found in under 1 s are plotted at 1 s.

Mutation Strategy Ablation. To evaluate each mutation strategy in isolation on TensorFlow v2.20, we considered three configurations of `FuzzWarp`: dimension-product overflow, boundary-value, and zero-dimension mutations. In each configuration, all other mutation strategies were disabled. We conducted one five-seed campaign for each configuration, using the same per-kernel time budget as the main campaign. Table VI reports the union of unique GPU memory errors found across the five seeds of each campaign. The dimension-product overflow strategy uncovered 25 unique errors, while the boundary-value and zero-dimension strategies uncovered 11 and 9, respectively. Across the three ablation campaigns, the isolated strategies uncovered 39 unique errors. These campaigns were conducted separately from the final campaign above; their union is therefore not the 30 total errors of that campaign. Each mutation family uncovered bugs that the other two did not, showing that all three contribute to bug discovery.

Synchronization Impact. To assess the importance of per-mutation synchronization for reliable detection and correct mutation attribution, we re-ran `FuzzWarp` on TensorFlow v2.20 without synchronization while fuzzing only the 30 vulnerable kernels (under the same per-kernel time budget). In this configuration, only 17 of the 30 GPU memory errors (57%) were re-discovered, indicating that *omitting synchronization leads to error misattribution and missed bugs*.

TABLE V
FUZZED TARGETS AND GPU MEMORY ERRORS IN TENSORFLOW (V2.20) AND PYTORCH (V2.13).

	TensorFlow Kernels		PyTorch CUDA Ops	
Run	**Fuzzed**	**GPU Errors**	**Fuzzed**	**GPU Errors**
1	265	15	482	40
2	265	16	467	42
3	266	19	667	46
4	266	21	524	36
5	266	18	478	36
Total (unique)	**267**	**30**	**708**	**69**

TABLE VI
MUTATION-STRATEGY ABLATION ON TENSORFLOW V2.20. EACH ROW REPORTS UNIQUE GPU MEMORY ERRORS ACROSS FIVE SEEDS FROM ONE INDEPENDENT CAMPAIGN, WITH ONLY THE INDICATED STRATEGY ENABLED.

Mutation Strategy	**Seeds**	**Unique Errors**
Dimension-product Overflow	5	25
Boundary-value	5	11
Zero-dimension	5	9
Unique union	**–**	**39**

Sanitizer Impact. To evaluate whether native CUDA error reporting is sufficient to validate the reproducibility of the synthesized PoVs, we re-executed all 30 TensorFlow PoVs under configurations that omit Compute Sanitizer: (i) with synchronization and CUDA error checking enabled, and (ii) with explicit synchronization disabled. In both cases, 7 out of 30 PoVs (23%) did not reliably surface a CUDA fault, despite being reproducible under Compute Sanitizer. This indicates that *a non-trivial fraction of GPU memory errors can remain silent* with respect to CUDA's error reporting (e.g., no host-visible error), motivating the use of Compute Sanitizer for checking device-side memory accesses to avoid false negatives.

D. Vulnerability Analysis

Table VII presents a taxonomy of the root causes identified across the 30 TensorFlow and 69 PyTorch vulnerabilities. For TensorFlow, integer overflow is the most common root cause, accounting for 77% of cases. Although there are cases where this happens because values are stored incorrectly in `int32` variables that are later used for some operation, this prevalence is largely driven by the pervasive use of the `GPU_1D_KERNEL_LOOP` macro (see Listing 7) [71], which enforces signed 32-bit integer indexing for CUDA kernels. This macro, defined as '`for (int i : GpuGridRangeX<int>(n))`', *explicitly truncates* loop counters to 32-bit `int` types. Consequently, when intermediate stride calculations exceed the bounds of a 32-bit signed

```
// tensorflow/core/util/gpu_kernel_helper.h
#define tf_tensorflow
#define GPU_1D_KERNEL_LOOP(i, n) \
  for (int i : ::tf::GpuGridRangeX<int>(n))

template <typename T>
__device__ detail::GpuGridRange<T>
  GpuGridRangeX(T count) {
    return detail::GpuGridRange<T>(
  /*begin=*/blockIdx.x * blockDim.x + threadIdx.x,
  /*delta=*/gridDim.x * blockDim.x,
  /*end=*/count);
}
```

Listing 7: TensorFlow utility macro and helper (`GPU_1D_KERNEL_LOOP`) used for 1D grid-stride loops.

integer, the values wrap around to negative numbers. If these overflowed values are then used as offsets in memory access operations (e.g., `ptr[index * stride]`), they may lead to *heap out-of-bounds* memory load/store operations at massive negative offsets relative to an allocated object/buffer.

The remaining TensorFlow vulnerabilities are categorized as direct *out-of-bounds accesses* (20%), where kernels fail to validate user-supplied indices against object/buffer bounds, and *NULL pointer dereferences* (3%), arising from edge cases involving zero-dimension tensors.

For PyTorch, out-of-bounds access is the most common root cause, accounting for 75% of cases. Most of these errors occur because CUDA kernels use tensor dimensions, indices, or auxiliary metadata to determine memory accesses without checking that the referenced buffers contain the corresponding elements. The remaining vulnerabilities are misaligned accesses (13%), where vectorized kernels access pointers that do not meet alignment requirements, and integer overflows (12%), where signed 32-bit calculations wrap or truncate.

From an exploitation perspective, Table VIII shows that most vulnerabilities enable *relative read/write primitives*. For TensorFlow, 25 vulnerabilities (83.3%) enable relative read/write primitives, three (10.0%) provide *arbitrary* heap read/write capabilities, one (3.3%) provides an arbitrary shared-memory write, and one (3.3%) causes denial of service. For PyTorch, 58 vulnerabilities (84.1%) enable relative read/write primitives, two (2.9%) provide arbitrary read/write capabilities, and nine (13.0%) cause denial of service. Such primitives could be leveraged to leak model weights, corrupt inference results, perform privilege escalation on CPU [13], or potentially achieve code execution in multi-tenant GPU settings [12], [34].

E. Vulnerability Abuse in Model-serving Workflows

To move beyond direct kernel-level testing, we construct an end-to-end experiment around an ImageNet-pretrained VGG-11 classifier [72], and evaluate whether a `FuzzWarp`-identified vulnerability can be triggered during model inference. We deploy the model through Triton, and implement a client that submits valid JPEG images via its HTTP inference interface. We use the published KerasHub `vgg_11_imagenet` preset [73], which preserves input dimensions by default, without modifying the model architecture, pretrained weights, or preprocessing configuration. When the client sends a JPEG image of size $44{,}561 \times 753$ pixels, the first convolution produces a float32 tensor containing 2,147,483,712 elements, exceeding `INT32_MAX`. TensorFlow's internal conversion from NCHW to NHWC reaches the vulnerable `SwapDimension1And2InTensor3UsingTiles` kernel, where signed 32-bit index calculations overflow and produce out-of-bounds writes. This experiment demonstrates that the vulnerability is reachable through an input to an unchanged pretrained model, rather than only through direct operation invocation.

TABLE VII
ROOT-CAUSE CLASSIFICATION OF THE UNCOVERED BUGS.

	TensorFlow		PyTorch	
Category	**Count**	**(%)**	**Count**	**(%)**
Integer overflow	23	76.7	8	11.6
Out-of-bounds access	6	20.0	52	75.4
`NULL` pointer dereference	1	3.3	0	0.0
Misaligned access	0	0.0	9	13.0
Total	**30**	**100.0**	**69**	**100.0**

TABLE VIII
SECURITY IMPACT ANALYSIS OF THE UNCOVERED BUGS.

Capability	**Exploitation Primitive**	**TF (%)**	**PT (%)**
Heap OOB Write	Arbitrary write	2 (6.7)	1 (1.4)
	Relative write (pre-buffer/object)	3 (10.0)	4 (5.8)
	Relative write (post-buffer/object)	6 (20.0)	7 (10.1)
Heap OOB Read	Arbitrary read	1 (3.3)	1 (1.4)
	Relative read (pre-buffer/object)	7 (23.3)	6 (8.7)
	Relative read (post-buffer/object)	9 (30.0)	41 (59.4)
Shared-memory OOB	Arbitrary shared-memory write	1 (3.3)	0 (0.0)
Denial of Service	Crash	1 (3.3)	9 (13.0)
Total		**30 (100.0)**	**69 (100.0)**

X. CONCLUSION

We presented `FuzzWarp`, a GPU-aware fuzzer that is specifically designed to detect memory-safety vulnerabilities in DL-framework kernels executing on GPUs. Unlike prior work that primarily targets CPU-side errors or operates at high(er)-level API layers, `FuzzWarp` adopts a bottom-up approach that directly instruments and fuzzes GPU kernel implementations. Our approach addresses three fundamental challenges unique to GPU fuzzing: ensuring that execution reaches GPU-specific code paths rather than CPU fallbacks; maintaining correct device placement during tensor mutation; and reliably detecting asynchronous GPU memory errors that do not manifest as synchronous process crashes. Our evaluation on multiple TensorFlow versions demonstrates that `FuzzWarp` consistently outperforms state-of-the-art DL fuzzers. Across TensorFlow v2.20 and PyTorch v2.13, `FuzzWarp` discovered 99 GPU memory errors.

Availability

The FuzzWarp prototype is available at: https://gitlab.com/brown-ssl/fuzzwarp. Our evaluated artifact can be found at: https://doi.org/10.5281/zenodo.22963932.

Acknowledgment

We thank our anonymous shepherd and reviewers for their valuable feedback. This work was supported by the National Science Foundation (NSF), through awards 2238467, 2526622, and 2247562. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the US government or NSF.

Ethics Considerations

Stakeholders. Our research involves and impacts several key groups of stakeholders. First, the developers of DL frameworks (e.g., the maintainers of TensorFlow and PyTorch) are directly affected, as our work identifies security vulnerabilities in their codebase that require remediation. Second, application developers and organizations deploying DL framework code in production, specifically in safety-critical domains, such as autonomous driving, healthcare, and industrial control, rely on the memory safety of the underlying GPU-kernel code. Third, the end-users of these AI-powered applications are indirect stakeholders who face potential privacy or safety risks if the underlying infrastructure is compromised. At the same time, security researchers and practitioners benefit from our methodological advancements in fuzzing GPU-kernel code of DL frameworks, as well as from the disclosure of new vulnerabilities, their accompanying root-cause analysis and attribution, and exploitability investigation.

Impact. We consider both the positive security implications of our work and the potential risks arising from its misuse:

(a) Ethical Principles: We are guided by the principles of Beneficence and Respect for Law and Public Interest as outlined in the Menlo Report [74]. Our objective is to strengthen the security of the computational infrastructure underpinning modern AI. By proactively identifying and reporting vulnerabilities, we aim to prevent exploitation that could compromise the integrity of critical systems, which rely on DL-framework code, or lead to privilege escalation and lateral movement [13].

(b) Potential Harms: The primary risk of our work is the potential for dual-use. FuzzWarp and the synthesized PoV snippets can be misused by malicious actors to identify zero-day vulnerabilities or construct exploits for unpatched systems. Specifically, memory-safety vulnerabilities in GPU code can be leveraged for denial-of-service attacks, unauthorized data exfiltration, silent data corruption in ML models, or even to compromise the host system's security guarantees.

Mitigations. We have taken several steps to mitigate the aforementioned risks:

1. Responsible Disclosure: We followed the respective community guidelines and reported 35 TensorFlow and 53 PyTorch PoVs, including ones from earlier development runs. For PyTorch, we reported only PoVs that use APIs the maintainers consider public. The other PoVs use internal APIs that are still public-facing, since any developer can call them, but the maintainers object to reports that rely on them. The PyTorch maintainers acknowledged 47 of the 53; the other six were reported last and are still awaiting triage. Of the 47, 10 are fixed, while 36 were classified as duplicates of one umbrella issue involving missing shape checks (PyTorch issue #195547)—one of these has since been fixed. For TensorFlow, all 35 PoVs were acknowledged, and 34 have merged or approved fixes. Of the 99 bugs in Section IX-C, 37 have such fixes. Developer confirmation refers to maintainer acknowledgment of the reported fault and its reproducer; it does not imply that a patch was merged or a CVE was assigned.

2. Controlled Experimentation: All fuzzing experiments were conducted in local, isolated environments. No production systems were tested, ensuring no disruption to external stakeholders during the research process.

3. Responsible Release: FuzzWarp's source code is public on GitLab (https://gitlab.com/brown-ssl/fuzzwarp); the complete artifact, including the TensorFlow PoVs and evaluation scripts, is archived on Zenodo (https://doi.org/10.5281/zenodo.22963932—see Appendix A).

4. Human Subjects: This research focused solely on software and hardware artifacts. It did not involve any human participants nor the collection of personal data; therefore, no IRB approval was required.

Decision to Conduct and Publish. We carefully weighed the potential risks of dual-use against the benefits of our research. We concluded that the vulnerabilities in GPU-kernel code of DL frameworks, like TensorFlow, exist independently of our study and that leaving them undiscovered poses a greater risk to the public interest than the risks associated with disclosing our findings. By identifying these flaws and coordinating with vendors to fix them, we actively contribute to the defense of the AI ecosystem. Thus, we determined that proceeding with the research and publication, under the framework of responsible disclosure, was the ethical course of action to support the principles of Beneficence and Respect for Law and Public Interest.

References

[1] A. Esteva, B. Kuprel, R. A. Novoa, J. Ko, S. M. Swetter, H. M. Blau, and S. Thrun, "Dermatologist-level classification of skin cancer with deep neural networks," *Nature*, vol. 542, no. 7639, pp. 115–118, 2017.

[2] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba, "End to End Learning for Self-Driving Cars," *Computing Research Repository (CoRR)*, vol. abs/1604.07316, 2016.

[3] P. Covington, J. Adams, and E. Sargin, "Deep Neural Networks for YouTube Recommendations," in *ACM Conference on Recommender Systems (RecSys)*, 2016, pp. 191–198.

[4] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention Is All You Need," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2017, pp. 5998–6008.

[5] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: A System for Large-Scale Machine Learning," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016, pp. 265–283.

[6] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2019, pp. 8024–8035.

[7] NVIDIA Corporation, "CUDA C++ Programming Guide: v13.0," 2025. [Online]. Available: https://docs.nvidia.com/cuda/archive/13.0.0/cuda-c-programming-guide/index.html

[8] D. Bang, J. Sun, and H. Chen, "A Study of Overflow Vulnerabilities on GPUs," in *IFIP International Conference on Network and Parallel Computing (NPC)*, 2016, pp. 103–115.

[9] A. Miele, "Buffer overflow vulnerabilities in CUDA: a preliminary analysis," *Journal of Computer Virology and Hacking Techniques*, vol. 12, no. 2, pp. 113–120, 2016.

[10] H. Naghibijouybari, A. Neupane, Z. Qian, and N. Abu-Ghazaleh, "Rendered Insecure: GPU Side Channel Attacks are Practical," in *ACM Conference on Computer and Communications Security (CCS)*, 2018, pp. 2139–2153.

[11] N. Lungu, S. S. Patra, M. R. Mishra, B. B. Dash, G. C. Sasmal, P. Pattnayak, S. Singh, and M. K. Gourisaria, "GPU Side-Channel Attack Classification for Targeted Secure Shader Mitigation," *SN Computer Science*, vol. 5, no. 8, pp. 1–23, 2024.

[12] Y. Guo, Z. Zhang, and J. Yang, "GPU Memory Exploitation for Fun and Profit," in *USENIX Security Symposium (USENIX Security)*, 2024, pp. 4033–4050.

[13] S. Roh, W. Choi, J. Chung, Y. Lee, S. Song, and B. Lee, "GHost in the Shell: A GPU-to-Host Memory Attack and Its Mitigation," in *IEEE Symposium on Security and Privacy (S&P)*, 2026, pp. 456–471.

[14] M. He, S. Xia, H. Jia, W. Wu, and L. Song, "M2K: Making the Model-Kernel Interface Explicit for Reliable CUDA Kernel Verification," in *ACM Symposium on Operating Systems Principles (SOSP)*, 2026, pp. 1091–1106.

[15] J. Roels, A. Jacobs, S. Vlasceanu, M. Ammar, and S. Volckaert, "From Prompt to Pwn: Exploiting GPU Memory Errors During ML Inference," in *ACM Conference on Computer and Communications Security (CCS)*, 2026.

[16] J. Lee, Y. Kim, J. Cao, E. Kim, J. Lee, and H. Kim, "Securing GPU via Region-based Bounds Checking," in *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2022, pp. 27–41.

[17] M. Pavlidakis, G. Vasiliadis, S. Mavridis, A. Argyros, A. Chazapis, and A. Bilas, "Guardian: Safe GPU Sharing in Multi-Tenant Environments," in *ACM/IFIP International Middleware Conference (Middleware)*, 2024, pp. 313–326.

[18] J. Lee, E. Chung, S. Singh, S. Na, Y. Kim, J. Lee, and H. Kim, "Let-Me-In: (Still) Employing In-pointer Bounds Metadata for Fine-grained GPU Memory Safety," in *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2025, pp. 1648–1661.

[19] M. Tarek Ibn Ziad, S. Damani, A. Jaleel, S. W. Keckler, and M. Stephenson, "cuCatch: A Debugging Tool for Efficiently Catching Memory Safety Violations in CUDA Applications," *Proceedings of the ACM on Programming Languages (PACMPL)*, vol. 7, no. PLDI, pp. 124–147, 2023.

[20] H. Lu, F. Zhang, Z. Zhang, S. Wang, and Y. Guo, "CuSafe: Capturing Memory Corruption on NVIDIA GPUs," in *USENIX Security Symposium (USENIX Security)*, 2026, pp. 2745–2762.

[21] NVIDIA Corporation, "NVIDIA Compute Sanitizer Documentation, v2025.3.1," 2025. [Online]. Available: https://docs.nvidia.com/compute-sanitizer/ComputeSanitizer/index.html

[22] T. M. Baumann and J. Gracia, "Cudagrind: A Valgrind Extension for CUDA," in *Parallel Computing: Accelerating Computational Science and Engineering (CSE)*. IOS Press, 2014, pp. 763–772.

[23] N. Christou, D. Jin, V. Atlidakis, B. Ray, and V. P. Kemerlis, "IvySyn: Automated Vulnerability Discovery in Deep Learning Frameworks," in *USENIX Security Symposium (USENIX Security)*, 2023, pp. 2383–2400.

[24] C. Peng and A. Rajan, "Automated Test Generation for OpenCL Kernels Using Fuzzing and Constraint Solving," in *Workshop on General Purpose Processing Using GPUs (GPGPU)*, 2020, pp. 61–70.

[25] X. Zou, J. Pu, A. Srivastav, J. Cox, Z. Liang, Y. Tan, X. Li, Y. Zhu, and Z. Qian, "StepStone: LLM-Based GPU Kernel Driver Fuzzing via User-Space Libraries," in *IEEE Symposium on Security and Privacy (S&P)*, 2026, pp. 1–18.

[26] NVIDIA Corporation, "Dynamo-Triton," NVIDIA Developer. [Online]. Available: https://developer.nvidia.com/dynamo-triton

[27] C++ reference, "Function template." [Online]. Available: https://en.cppreference.com/w/cpp/language/function_template.html

[28] PyTorch, "Custom C++ and CUDA Operators," 2026. [Online]. Available: https://docs.pytorch.org/tutorials/advanced/cpp_custom_ops.html#registering-backend-implementations-for-an-operator

[29] TensorFlow, "Create an op," 2024. [Online]. Available: https://www.tensorflow.org/guide/create_op

[30] Advanced Micro Devices, Inc., "TensorFlow compatibility," ROCm Documentation, 2026. [Online]. Available: https://rocm.docs.amd.com/en/docs-7.2.1/compatibility/ml-compatibility/tensorflow-compatibility.html

[31] PyTorch, "PyTorch bindings." [Online]. Available: https://github.com/pytorch/pytorch/blob/master/tools/autograd/gen_python_functions.py

[32] TensorFlow, "Keras: The high-level API for TensorFlow," 2023. [Online]. Available: https://www.tensorflow.org/guide/keras

[33] W.-m. W. Hwu, D. B. Kirk, and I. El Hajj, *Programming Massively Parallel Processors: A Hands-on Approach*, 4th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2022.

[34] J. Roels, A. Jacobs, and S. Volckaert, "CUDA, Woulda, Shoulda: Returning Exploits in a SASS-y World," in *ACM European Workshop on Systems Security (EuroSec)*, 2025, pp. 40–48.

[35] A. One, "Smashing The Stack For Fun And Profit," *Phrack Magazine*, vol. 7, no. 49, 1996.

[36] H. Shacham, "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)," in *ACM Conference on Computer and Communications Security (CCS)*, 2007, pp. 552–561.

[37] G. S. Kc, A. D. Keromytis, and V. Prevelakis, "Countering Code-Injection Attacks With Instruction-Set Randomization," in *ACM Conference on Computer and Communications Security (CCS)*, 2003, pp. 272–280.

[38] K. S. Yim, C. Pham, M. Saleheen, Z. Kalbarczyk, and R. Iyer, "Hauberk: Lightweight Silent Data Corruption Error Detector for GPGPU," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2011, pp. 287–300.

[39] H. Wu, Q. Zhao, S. Chen, Y. Chen, Y. Hao, T. C. Liu, S. Chen, A. Aziz, and K. Zhou, "Triton-Sanitizer: A Fast and Device-Agnostic Memory Sanitizer for Triton with Rich Diagnostic Context," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2026, pp. 2108–2124.

[40] J. Shi, Y. Xiao, Y. Li, Y. Li, D. Yu, C. Yu, H. Su, Y. Chen, and W. Huo, "ACETest: Automated Constraint Extraction for Testing Deep Learning Operators," in *ACM International Symposium on Software Testing and Analysis (ISSTA)*, 2023, pp. 690–702.

[41] D. Xie, J. Wang, H. V. Pham, L. Tan, Y. Guo, A. Aziz, and E. Meijer, "CEDAR: Continuous Testing of Deep Learning Libraries," in *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2024, pp. 371–382.

[42] D. Xie, Y. Li, M. Kim, H. V. Pham, L. Tan, X. Zhang, and M. W. Godfrey, "DocTer: Documentation-Guided Fuzzing for Testing Deep Learning API Functions," in *ACM International Symposium on Software Testing and Analysis (ISSTA)*, 2022, pp. 176–188.

[43] N. Shiri Harzevili, M. M. Mohajer, M. Wei, H. V. Pham, and S. Wang, "History-Driven Fuzzing for Deep Learning Libraries," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 34, no. 1, pp. 1–29, 2025.

[44] Z. Deng, G. Meng, K. Chen, T. Liu, L. Xiang, and C. Chen, "Differential Testing of Cross Deep Learning Framework APIs: Revealing Inconsistencies and Vulnerabilities," in *USENIX Security Symposium (USENIX Security)*, 2023, pp. 7393–7410.

[45] H. J. Kang, P. Rattanukul, S. A. Haryono, T. G. Nguyen, C. Ragkhitwetsagul, C. Pasareanu, and D. Lo, "SkipFuzz: Active Learning-based Input Selection for Fuzzing Deep Learning Libraries," *Computing Research Repository (CoRR)*, vol. abs/2212.04038, 2022.

[46] K. Zhang, S. Wang, J. Han, X. Zhu, X. Li, S. Wang, and S. Wen, "Your Fix Is My Exploit: Enabling Comprehensive DL Library API Fuzzing with Large Language Models," in *IEEE/ACM International Conference on Software Engineering (ICSE)*, 2025, pp. 3110–3122.

[47] S. Kim, Y. Kim, D. Park, Y. Jeon, J. Yi, and M. Kim, "Lightweight Concolic Testing via Path-Condition Synthesis for Deep Learning Libraries," in *IEEE/ACM International Conference on Software Engineering (ICSE)*, 2025, pp. 2957–2969.

[48] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, "Large Language Models Are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via

Large Language Models," in *ACM International Symposium on Software Testing and Analysis (ISSTA)*, 2023, pp. 423–435.
[49] G. Go, C. Zhou, Q. Zhang, X. Zou, H. Shi, and Y. Jiang, "Towards More Complete Constraints for Deep Learning Library Testing via Complementary Set Guided Refinement," in *ACM International Symposium on Software Testing and Analysis (ISSTA)*, 2024, pp. 1338–1350.
[50] A. Wei, Y. Deng, C. Yang, and L. Zhang, "Free Lunch for Testing: Fuzzing Deep-Learning Libraries from Open Source," in *IEEE/ACM International Conference on Software Engineering (ICSE)*, 2022, pp. 995–1007.
[51] C. Yang, Y. Deng, J. Yao, Y. Tu, H. Li, and L. Zhang, "Fuzzing Automatic Differentiation in Deep-Learning Libraries," in *IEEE/ACM International Conference on Software Engineering (ICSE)*, 2023, pp. 1174–1186.
[52] Y. Deng, C. Yang, A. Wei, and L. Zhang, "Fuzzing Deep-Learning Libraries via Automated Relational API Inference," in *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2022, pp. 44–56.
[53] J. Liu, J. Peng, Y. Wang, and L. Zhang, "NeuRI: Diversifying DNN Generation via Inductive Rule Inference," in *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2023, pp. 657–669.
[54] Z. Liu, Q. Zou, T. Yu, X. Wang, G. Meng, K. Chen, and D. Zhang, "ConFL: Constraint-guided Fuzzing for Machine Learning Framework," *Computing Research Repository (CoRR)*, vol. abs/2307.05642, 2023.
[55] J. Wang, T. Lutellier, S. Qian, H. V. Pham, and L. Tan, "EAGLE: Creating Equivalent Graphs to Test Deep Learning Libraries," in *IEEE/ACM International Conference on Software Engineering (ICSE)*, 2022, pp. 798–810.
[56] P. Ji, Y. Feng, D. Wu, L. Yan, P. Chen, J. Liu, and Z. Zhao, "MoCo: Fuzzing Deep Learning Libraries via Assembling Code," *IEEE Transactions on Software Engineering (TSE)*, vol. 51, no. 2, pp. 371–388, 2025.
[57] Z. Wang, M. Yan, J. Chen, S. Liu, and D. Zhang, "Deep Learning Library Testing via Effective Model Generation," in *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2020, pp. 788–799.
[58] J. Gu, X. Luo, Y. Zhou, and X. Wang, "Muffin: Testing Deep Learning Libraries via Neural Architecture Fuzzing," in *IEEE/ACM International Conference on Software Engineering (ICSE)*, 2022, pp. 1418–1430.
[59] Y. Deng, C. S. Xia, C. Yang, S. D. Zhang, S. Yang, and L. Zhang, "Large Language Models are Edge-Case Generators: Crafting Unusual Programs for Fuzzing Deep Learning Libraries," in *IEEE/ACM International Conference on Software Engineering (ICSE)*, 2024, pp. 1–13.
[60] M. Li, D. Li, J. Liu, J. Cao, Y. Tian, and S.-C. Cheung, "Enhancing Differential Testing with LLMs for Testing Deep Learning Libraries," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 35, no. 4, pp. 1–39, 2026.
[61] K. Zhang, D. Xiao, D. Wu, S. Wang, J. Zhao, Y. Lin, T. Xu, and S. Wang, "LLM-Powered Silent Bug Fuzzing in Deep Learning Libraries via Versatile and Controlled Bug Transfer," *Proceedings of the ACM on Programming Languages (PACMPL)*, vol. 10, no. OOPSLA1, pp. 1599–1626, 2026.
[62] Z. Li, H. Lu, Y. Guo, Z. Zhang, S. Wang, and F. Zhang, "GPU-Fuzz: Finding Memory Errors in Deep Learning Frameworks," in *ACM/IEEE Design Automation Conference (DAC)*, 2026.
[63] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "SoftBound: Highly Compatible and Complete Spatial Memory Safety for C," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009, pp. 245–258.
[64] ——, "CETS: Compiler-Enforced Temporal Safety for C," in *ACM SIGPLAN International Symposium on Memory Management (ISMM)*, 2010, pp. 31–40.
[65] IBM Corporation, "IBM Watson Discovery," IBM Products. [Online]. Available: https://www.ibm.com/products/watson-discovery
[66] A. K. Kamath and A. Basu, "iGUARD: In-GPU Advanced Race Detection," in *ACM Symposium on Operating Systems Principles (SOSP)*, 2021, pp. 49–65.
[67] J. Jacobson, M. Burtscher, and G. Gopalakrishnan, "HiRace: Accurate and Fast Data Race Checking for GPU Programs," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2024, pp. 1–14.
[68] NVIDIA Corporation, "NVIDIA RTX A4000 Datasheet," 2022. [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/gtcs21/rtx-a4000/nvidia-rtx-a4000-datasheet.pdf
[69] ——, "NVIDIA A40 Datasheet," 2022. [Online]. Available: https://images.nvidia.com/content/Solutions/data-center/a40/nvidia-a40-datasheet.pdf
[70] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating Fuzz Testing," in *ACM Conference on Computer and Communications Security (CCS)*, 2018, pp. 2123–2138.
[71] TensorFlow Authors, "TensorFlow: GPU_1D_KERNEL_LOOP Macro Definition," 2026. [Online]. Available: https://github.com/tensorflow/tensorflow/blob/b67ec509694f91188cd2ba9d8e818952e984e58f/tensorflow/core/util/gpu_kernel_helper.h#L37
[72] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," in *International Conference on Learning Representations (ICLR)*, 2015.
[73] Keras, "VGG-11 Pretrained on ImageNet (vgg_11_imagenet), Version 2." [Online]. Available: https://www.kaggle.com/models/keras/vgg/keras/vgg_11_imagenet/2
[74] M. Bailey, D. Dittrich, E. Kenneally, and D. Maughan, "The Menlo Report," *IEEE Security & Privacy (S&P)*, vol. 10, no. 2, pp. 71–75, 2012.

APPENDIX A
ARTIFACT APPENDIX

This artifact contains `FuzzWarp`, the TensorFlow PoVs reported in the paper, the four comparison-fuzzer environments, and scripts for reproducing the GPU-oracle, ablation, and Triton experiments. The main entry point is `artifacts_evaluation/REVIEWER_REPRODUCTION.md`.

A. Description & Requirements

1) How to access: The artifact is publicly available on Zenodo at https://doi.org/10.5281/zenodo.22963932. It contains the `FuzzWarp` source ZIP, four compressed Docker images, `SHA256SUMS`, and `IMAGE_MANIFEST.txt`.

2) Hardware dependencies: Running the GPU experiments requires an x86-64 Linux host and an NVIDIA GPU. The comparison images regarding Section IX-B require compute capability 8.x with minor version 6 or newer and at least 16GB of VRAM; we tested them on an RTX A4000 and A40 (compute capability 8.6), and independent artifact evaluation passed on an L40S (8.9). The `FuzzWarp` experiments in Section IX-C require compute capability 8.0 or newer and were tested on an A30 (8.0, 24GB), A40 (8.6), and independently on an L40S (8.9). Running E5 requires a dedicated NVIDIA GPU. Plan for at least 250GB of disk space.

3) Software dependencies: The host needs Bash v5 or newer, Python v3, GNU Coreutils, `curl`, `unzip`, `zstd`, Docker Engine v20.10 or newer, NVIDIA Container Toolkit, and NVIDIA driver v560 or newer. A host CUDA Toolkit is not required. Internet access is needed for the first `FuzzWarp` TensorFlow and Triton builds. We tested the launchers with Docker v24.0.7 and NVIDIA Container Toolkit v1.16.1; the independent comparison smoke test used Docker v29.2.1.

4) Benchmarks: The artifact contains the TensorFlow v2.20 kernel-test workload used by `FuzzWarp`, the PoVs from campaign seeds 1–5, and pinned images for ACETest, TensorScope, Orion, and TitanFuzz. E5 reruns the VGG-11 inference request through Triton under Compute Sanitizer. A full TitanFuzz campaign also downloads `facebook/incoder-1B`.

B. Artifact Installation & Configuration

Download all Zenodo files into one directory, verify their checksums with `sha256sum -c SHA256SUMS`, and extract the source ZIP. Then run `chmod -R a+rX fuzzwarp-source`, which is required when the ZIP was extracted under a restrictive umask. Decompress each `*.tar.zst` file into `docker load`, then run the environment check from the source root. The exact commands and expected image tags are in `REVIEWER_REPRODUCTION.md`. The `FuzzWarp` smoke test builds TensorFlow and is machine- and network-dependent; later runs reuse the build cache.

C. Experiment Workflow

Start with the environment check. Next, run the four short ON/OFF comparison checks and the one-kernel `FuzzWarp` smoke test. For an evaluation that fits in one day, also run the three-hour-bounded sanitizer ablation, run the end-to-end VGG-11 Triton reproduction for serving reachability, and inspect the supplied synchronization PoVs and manifests. Each wrapper prints a timestamped result directory containing its environment, logs, generated PoVs, and summaries. Preserve that directory before rerunning the same launcher.

The complete five-seed TensorFlow `FuzzWarp` campaign and full synchronization experiment are included for exact methodological reproduction, but are not part of the one-day path. They take about 3.5 days and 10.5 hours, respectively, excluding setup.

D. Major Claims

- **(C1)** `FuzzWarp` found more unique GPU memory errors than each baseline in the four matched comparisons (Table II). The oracle experiment also found seven bugs that `FuzzWarp` missed in the matched campaigns: four from ACETest, one from Orion, and two from TensorScope. In the targeted E1 check, ACETest finds no correctly attributed out-of-bounds access with the oracle OFF and finds the triggering case with it ON (§IX-B).
- **(C2)** `FuzzWarp` reaches GPU kernels, finds allocator-bound GPU memory errors, and produces Python PoVs. Across five TensorFlow v2.20 seeds, the paper reports 267 kernels tested and 30 unique errors (E2; §IX-C).
- **(C3)** Without synchronization, `FuzzWarp` rediscovered 17 of the 30 vulnerable kernels. Without Compute Sanitizer, 7 of 30 PoVs did not reliably surface a host-visible GPU fault (E3; §IX-C).
- **(C4)** For TensorFlow, integer overflow accounts for 77% of the 30 findings (Table VII), and three findings provide arbitrary read/write capabilities (Table VIII)—E4; §IX-D.
- **(C5)** A TensorFlow GPU out-of-bounds access is reachable via a Triton model-serving workflow (E5; §IX-E).

E. Evaluation

1) Experiment (E1): GPU-oracle comparison: [About 12 compute-minutes after setup.] From the source root, run the comparison smoke wrapper under `artifacts_evaluation/` with `-gpu 0`. It checks ACETest, TensorScope, Orion, and TitanFuzz with the same testcase under oracle ON/OFF. ON must report the GPU fault at the triggering testcase; OFF misses it or reports it later under the wrong testcase. The wrapper reports PASS/FAIL for each fuzzer. The five-run launchers and preserved Table II PoVs are also included.

2) Experiment (E2): FuzzWarp smoke test: Run `bash artifacts_evaluation/run_fuzzwarp_smoke.sh` with `-gpu 0`. The initial TensorFlow build can take several hours; a cached run takes minutes. Success ends with `FuzzWarp smoke PASS` and produces a `DenseBincount_*.log` in which Compute Sanitizer reports an out-of-bounds global- or shared-memory access in any of the three vulnerable kernels documented in the DenseBincount report. This is a one-kernel end-to-end check, not a reproduction of the five-seed count. The complete campaign PoVs and canonical kernel list are in `povs_fuzzwarp_tf_2.20_campaign/`.

3) Experiment (E3): TensorFlow ablations: In `povs_fuzzwarp_tf_2.20_campaign/`, run `sanitizer_ablation/run.sh` with GPU ID 0. This replays the 30 PoVs without Compute Sanitizer, with and without explicit GPU synchronization. The run is bounded to three hours. On an A30, 23/30 PoVs reliably reported a host-visible fault and 7/30 did not in either mode. Read `summary.md`, `matrix.csv`, and `attempts.csv`. The environment report is written beside the timestamped results directory as `<timestamp>.environment.txt`. For the synchronization ablation, `synchronization_ablation/run.sh` reruns the fuzzer with synchronization disabled; count how many bugs it finds again. The full campaign is outside the one-day evaluation path, so inspect `synchronization_ablation/manifest.tsv` and `result_povs/`, which contains 17 PoVs for kernels in the paper's 30-kernel list. The saved sanitizer logs come from the final five-seed campaign.

4) Experiment (E4): Vulnerability analysis: Inspect the per-kernel reports in `tensorflow_vulnerability_analysis/` and compare them with the canonical 30-kernel list. These reports record the crashing kernel, access direction, fault address, allocation bounds, and root cause. The directory is a cumulative development-time PoV collection rather than the output of the final five-seed campaign. Its README maps the cumulative collection's operation-level entries to the final campaign's distinct CUDA-kernel findings.

5) Experiment (E5): Triton reachability: From the `FuzzWarp` source root, run the complete TensorFlow v2.20/JPEG experiment on physical GPU 0 with `bash artifacts_evaluation/run_triton_e5.sh 0`. The launcher builds the pinned Triton 2.55.0/TensorFlow 2.20.0 image, starts Triton under Compute Sanitizer, waits for VGG-11 to become ready, and sends the generated JPEG through its HTTP endpoint. A successful run prints `E5 PASS` and writes `results/triton-e5-TIMESTAMP/triton-sanitizer.txt`. The trace must contain 32 invalid global writes attributed to `SwapDimension1And2InTensor3UsingTiles`. The archived paper trace is `campaign_artifacts/vgg11_triton/triton/triton-sanitizer.txt`.