How Effective Is Coverage-Guided Fuzzing to Test Deep Learning Library APIs?

Feiran (Alex) Qin
NC State University
Raleigh, NC, USA
fqin2@ncsu.edu

M. M. Abid Naziri
NC State University
Raleigh, NC, USA
mnaziri@ncsu.edu

Hengyu Ai
ShanghaiTech University
Shanghai, China
aihy2023@shanghaitech.edu.cn

Saikat Dutta
Cornell University
Ithaca, NY, USA
saikatd@cornell.edu

Marcelo d'Amorim
NC State University
Raleigh, NC, USA
mdamori@ncsu.edu

***Abstract*—Deep Learning (DL) libraries (e.g., PyTorch) provide the core components to build major AI-enabled applications. Finding bugs in these libraries is an important and challenging problem. Prior approaches have tackled this challenge through either API-level fuzzing or model-level fuzzing. But, these approaches do not use coverage-guidance, which limits their effectiveness. This limitation raises an intriguing question: can coverage-guided fuzzing (CGF) be effectively applied to test DL libraries, and does it offer significant improvements in terms of standard metrics compared to prior methods?**

This paper reports on the first in-depth study to answer this question. We observe that a key challenge in applying popular CGF tools (e.g., libFuzzer) to test DL libraries is the need to synthesize a complete test harness, for each DL API, that can correctly transform byte-level inputs generated by the fuzzer into valid inputs that satisfy the complex input constraints of the API. To address this challenge and support this study, we propose FLASHFUZZ, a method that leverages Large Language Models (LLMs) to automatically synthesize API-level test harnesses. FLASHFUZZ combines harness templates, helper functions, and API documentation to synthesize harnesses, and employs a feedback-driven strategy to iteratively repair broken harnesses.

Our results show that FLASHFUZZ can successfully synthesize test harnesses for 1,271 and 786 APIs of PyTorch and TensorFlow, respectively. Compared to state-of-the-art fuzzing techniques for DL libraries (namely ACETEST, PATHFINDER, and TITANFUZZ), FLASHFUZZ achieves up to 212.88% higher coverage and 5.4x higher validity rate. Compared to these baselines, FLASHFUZZ achieves speedups of up to 1182x in generating inputs. Finally, FLASHFUZZ revealed 54 previously unknown bugs in the latest versions of PyTorch and TensorFlow, of which 53 bugs have been confirmed and 15 bugs have already been fixed by developers. Our study demonstrates that CGF is a strong baseline for API-level testing of DL libraries. We recommend future work to compare against CGF.

I. INTRODUCTION

Deep Learning (DL) libraries, such as TensorFlow [1] and PyTorch [2] are the backbone of modern machine learning applications. Like most complex software, these libraries also contain bugs [3], [4], [5], [6]. *Model-based fuzzing* [7], [8], [9], [10], [11], [12] and *API-level fuzzing* [5], [13], [14], [15], [16] are two complementary categories of bug finding techniques for DL library testing. Model-based fuzzing generates programs representing an ML model using the APIs from the DL library whereas API-level fuzzing focuses on generating inputs to individual APIs of the library. In this work, we focus on API-level fuzzing, which is more scalable and can be applied to a larger number of APIs, and has been shown to be effective at revealing many bugs.

Coverage-guided fuzzers (CGF), such as AFL++ [17] and libFuzzer [18], have uncovered bugs across diverse domains, including operating systems [19], compilers [20], and network protocols [21]. These fuzzers are known for their scalability, coverage-driven input generation, and low setup overhead. *However, their application to deep learning libraries remains largely unexplored. This paper reports on a study to assess the effectiveness of CGF for testing DL libraries.* This question is important given the limited scalability or coverage of existing techniques. For example, some techniques use expensive external tools, such as LLMs (e.g., TITANFUZZ [15]) and SMT solvers (e.g., ACETEST [22] and PATHFINDER [16]) to generate inputs, whereas some other techniques are oblivious to complex input constraints in the APIs (e.g., FREEFUZZ [5]), limiting their ability to cover deeper logic in code in a given time budget. We hypothesize that CGFs can explore the input spaces of DL APIs more efficiently to uncover bugs. In our work, we evaluate this hypothesis.

An important challenge in using CGF for API-level testing relates to scaling the synthesis of test harnesses to handle APIs with various signatures and input constraints. To address this challenge and support this study, we propose FLASHFUZZ, an Large Language Model (LLM)-based approach to automatically synthesize test harnesses. FLASHFUZZ (1) uses templates, helper functions, and API documentation to synthesize harnesses and (2) employs a feedback-directed mechanism to repair them and improve the synthesis success rate. Like recent API-level fuzzers (e.g., ACETEST [22] and PATHFINDER [16]), FLASHFUZZ builds on the observation that the majority of the important code of DL APIs are in the kernel functions, implemented in C++. So, FLASHFUZZ generates C++ test harnesses that directly invoke the C++-level DL APIs. Other LLM-based harness generation techniques exist (e.g., PromptFuzz [23] and CKGFuzzer [24]) but they are not specialized to DL APIs (See Section VII).

Method. FLASHFUZZ wraps libFuzzer [18] with an automated test harness synthesis approach. We evaluate FLASHFUZZ (i.e., the CGF method it represents) against three SoTA techniques using the standard metrics from the literature: coverage, validity ratio, and number of crashes revealed. We select ACETEST [22] and PATHFINDER [16] as they analyze

kernel implementations, as FLASHFUZZ does, and we select TITANFUZZ [15] as it represents a technique that uses Large Language Models (LLMs) to generate inputs. We conduct 8h fuzzing campaigns with the harnesses FLASHFUZZ generates to evaluate their ability to find new bugs.

Results. Our results show that FLASHFUZZ outperforms the comparison baselines on all metrics. In terms of coverage, FLASHFUZZ achieves 101.13%-212.88% higher coverage compared to the baselines across both DL libraries. Considering validity rate, FLASHFUZZ achieves 1.0x-1.3x higher validity rate for PyTorch and 1.7x-5.4x higher validity rate for TensorFlow compared to the baselines. Considering the ability to reveal crashes, we find that FLASHFUZZ finds more unique crashes than any other baseline. On PyTorch, FLASHFUZZ discovers 5, 5, and 16 more crashes than ACETEST, PATHFINDER, and TITANFUZZ, respectively. Only TITANFUZZ detects a single crash missed by FLASHFUZZ, while ACETEST and PATHFINDER find none. On TensorFlow, FLASHFUZZ discovers 18, 11, and 12 more crashes than ACETEST, PATHFINDER, and TITANFUZZ, respectively. We also conduct long-running fuzzing campaigns with FLASHFUZZ (8h per API) on the latest versions of PyTorch and TensorFlow to find bugs on their APIs. During these campaigns, FLASHFUZZ revealed a total of 54 new bugs; 32 in PyTorch and 22 in TensorFlow. Of these, developers fixed 15 so far.

Finally, this study presents key insights for the future of deep learning library testing. Our finding shows that CGF is an effective technique for testing DL APIs and should be used as a comparison baseline in future work. However, challenges remain in harness generation for complex APIs and triaging efforts. We discuss these lessons in Section VI-D.

This paper makes the following contributions:

- ⋆ **Study.** We report the first in-depth study of the use of coverage-guided fuzzing (CGF) to test DL APIs. To enable the study on a large number of APIs, we developed FLASHFUZZ, an LLM-based approach to automatically synthesize test harnesses for CGF.
- ⋆ **Bugs.** We report 54 new bugs in PyTorch and TensorFlow by applying FLASHFUZZ "in the wild", of which 15 have been fixed by developers.
- ⋆ **Artifacts.** We make our artifacts are publicly available [25].

II. BACKGROUND

A. Tensor Representation

A tensor is a data structure for representing and manipulating machine learning data in deep learning frameworks [26]. Tensors generalize vectors (first-order tensors) and matrices (second-order tensors) to arbitrary dimensions, providing a unified mathematical framework for deep learning computations [27], [28]. Tensors are characterized by four essential properties: (1) **Data Type** determines both the memory representation and the precision of computation. Common data types include floating-point (e.g., `float32`), integer (e.g., `int64`), and boolean values; (2) **Number of Dimensions (nDim)** defines the number of axes in the tensor; (3) **Shape** specifies the size of each dimension. For instance, a tensor with shape $(3, 4, 5)$ has 3 elements along its first dimension, 4 elements along its second dimension, and 5 elements along its third dimension. In total, the tensor contains 60 (=$3 \times 4 \times 5$) elements; (4) **Elements** constitute the individual values stored within the tensor. The total number of elements is the product of all dimension sizes.

B. Deep Learning Library Organization

Figure 1 shows the typical layered organization of a DL library. The kernel code, written in low-level languages like C++, implements the main functionalities of a library. A DL library provides different versions of a kernel to different devices, e.g., GPU kernel (for performance) and CPU kernel (for portability). Library users access these kernels through high-level APIs, often in Python or C++. Python and corresponding C++ APIs, in principle, implement the same "glue" functionality to access the kernel. This work tests the kernel code through C++ APIs, highlighted at the top right of the figure.

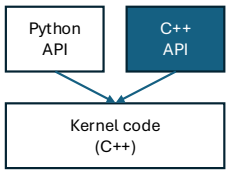


Fig. 1: Layered organization of a Deep Learning Library.

C. Coverage-Guided Fuzzing

Coverage-guided fuzzing (CGF) is a testing technique that uses coverage information to guide test input data generation [29], [18], [30]. CGF uses a queue to store input data, which is typically encoded as a byte array. In a given iteration, CGF dequeues an element from the queue, generates mutations for that input, translates the input byte array to the actual input data, and executes the target program on those inputs. The inputs that uncover new branches are added back to the queue for further processing. Fuzzing tools expect developers to instrument the program to be tested to collect coverage information (e.g., LLVM compiler option `-fsanitize=fuzzer-no-link`). Additionally, these tools expect developers to provide a *test harness* that (1) translates a byte array representation to the representation the program uses; (2) invokes the target program on that input; and (3) checks the correctness of the corresponding outputs.

Fuzzing is computationally intensive. To accelerate bug finding, fuzzing tools often use in-process fuzzing [18] where the harness and target are compiled together (see LLVM `-fsanitize=fuzzer`). This design enables the harness to call the target with function calls instead of system calls. We use libFuzzer as our fuzzing engine because it integrates with the LLVM compiler used by both PyTorch and TensorFlow, and its in-process fuzzing avoids the overhead of repeatedly initializing the DL framework. Testing from the Python frontend would require crossing the Python-to-C++ boundary, making coverage guidance impractical. Other engines (e.g., AFL++ [17], HonggFuzz [30]) could also be used.

III. ILLUSTRATIVE EXAMPLE

This section summarizes harness synthesis (§III-A), an example harness for `torch.fmod` (§III-B), and how CGF uses harnesses to test APIs (§III-C).

```cpp
#define MAX_RANK 4 // max number of tensor dimensions
#define MIN_RANK 0 // min number of tensor dimensions

extern "C" int LLVMFuzzerTestOneInput(const uint8_t* data, size_t size) {
 size_t offset = 0;
 tensorflow::Scope root = tensorflow::Scope::NewRootScope();
 try {
   // --- 1. Create Fuzzed Input Tensors ---
   // --- 2. Call the Target TensorFlow API ---
   // --- 3. Run the TensorFlow API ---
   if (!status.ok()) {
     std::cout<<"TF Exception:"<<status.ToString()<<std::endl;
     return -1; } // will discard the seeds
 } catch (const std::exception& e) {
   tf_fuzzer_utils::logError("TF Exception: " +
     std::string(e.what()), data, size);
   return -1;
 } // end catch
 return 0;
} // end try
```

Listing (1) Test harness template.

```cpp
tensorflow::DataType parseDataType(uint8_t selector) {
 tensorflow::DataType dtype;
 switch (selector % 20) {
   case 0:
     dtype = DT_FLOAT;
     break;
   // other 19 cases are omitted for brevity
 }
 return dtype;
}
uint8_t parseRank(uint8_t byte) {
 constexpr uint8_t range = MAX_RANK - MIN_RANK + 1;
 uint8_t rank = byte % range + MIN_RANK;
 return rank;
}
std::vector<int64_t> parseShape(const uint8_t* data, size_t& offset, size_t
      total_size, uint8_t rank) {...}

void createTensor(tensorflow::Tensor& tensor, const uint8_t* data, size_t& offset,
       size_t total_size) {...}
```

Listing (2) A subset of the helper functions used by FLASHFUZZ.

A. Test Harness Generation with FLASHFUZZ

FLASHFUZZ uses different artifacts to generate test harnesses that create a high number of semantically-valid inputs. First, FLASHFUZZ uses a C++ template that contains the overall structure of a test harness (Listing 1), including setup code for each framework, and checks for various exceptions. Second, FLASHFUZZ uses the API signature and documentation to guide the generation of code that parses raw input data and generates semantically-valid inputs. For example, for the TensorFlow API `acos` the documentation states *"Computes acos of parameter x element-wise... Input range is [-1, 1]... x: A Tensor. Must be one of the following types: bfloat16, ..."* [31]. Third, FLASHFUZZ defines and uses helper functions that parse raw data into objects of a certain type (Listing 2).

Figure 4 shows the prompt used to generate a test harness for `tf.raw_ops.Acos` [31]. The left side shows the prompt template specifying the target library, API name, documentation, library-specific harness template (Listing 1) and helper functions (Listing 2), and *requirements* instructing the LLM on tensor creation and API invocation. The right-hand side of Figure 4 shows a test harness for the API `fmod` that FLASHFUZZ generates using the prompt template shown on the left-hand side of the figure. This API takes only one input tensor as input but the parsing method FLASHFUZZ employs generalizes to APIs with arbitrary number and type of parameters. Note that the parameters in these libraries are mostly numeric types, container types (e.g., tensor, list, and tuple), and categorical values (e.g., boolean flags, string modes like `padding` = "`SAME`").

The test harness from Figure 4 (right) follows several steps to test the API. It parses the byte array `data` (steps 1-3), creates a tensor object `input_tensor` (step 4), calls the API `Acos` on the parsed inputs (step 5) with framework specific constructs (e.g., TensorFlow `Session`), and applies differential oracles to detect output divergences and exceptions between CPU and GPU executions (steps 6-7).

B. A test harness revealing a bug in PyTorch's `fmod`

We illustrate bug detection with `torch.fmod`, which involves richer parameter constraints and demonstrates a real bug found by FLASHFUZZ and fixed by developers. This API computes element-wise remainder of division, taking one tensor and either a scalar or another tensor. It supports 12 data types (e.g., `torch.float32`, `torch.int64`) and has two constraints: (1) if the second parameter is a tensor, it must match the first tensor's shape; (2) if it is a scalar, it must be non-zero. Fuzzing this API reveals a bug that we report to PyTorch developers, who confirmed and fixed the bug [32], [33]. The bug manifests when calling the API with a specific large negative number; the function crashes, raising a *"Floating point exception (core dumped)"* error on CPU, while the GPU implementation handles the input correctly. The root cause is undefined behavior in the CPU kernel due to signed integer overflow, which the developers fixed by adding a conditional check to align the CPU behavior with the GPU implementation.

C. CGF with a FLASHFUZZ Test Harnesses

Now we present how the test harness for `fmod` works with a CGF fuzzer and effectively generates semantically-valid inputs via mutation. Figure 5 shows the layout of a byte array representing an input for the API `torch.fmod` and its mutation process [34]. In step ❶, FLASHFUZZ reads the first byte of the byte array to determine the type of the tensor. The value `0x13` maps to 7 via a modulo 12 operation (FLASHFUZZ supports 12 PyTorch data types) and encodes the data type `torch.float32`. In step ❷, FLASHFUZZ extracts 2 as the number of dimensions in the tensor. It uses modulo 4 in that case as that is the default setting of FLASHFUZZ for number of dimensions. In step ❸, FLASHFUZZ reads the shape of the tensor from two sequences of four bytes. As before, FLASHFUZZ applies modulo 16 operations – as that is the bound on the maximum length of each dimension – to find that the tensor has a (3, 4) shape. Finally, in step ❹, FLASHFUZZ extracts the 12 ($=3\times4$) `float32` values (4 bytes each) from the byte array and arranges them according to the determined structure, creating a complete tensor with floating-point values from the rest of the input byte array.

Importantly, any mutation in the input byte array affects the tensor object to be created, as shown at the right-hand side of Figure 5. For example, let us suppose the fuzzer mutates the first byte (❶ in the input byte array) from `0x13` to `0x14` resulting in a change from `float32` to `int8`. In that case, FLASHFUZZ would consume 12 bytes ($=12\times1$) instead of 48 bytes ($=12\times4$) to define the elements of the 3×4 tensor. Consequently, the numeric elements of the generated

Prompt Template

Target Library: TensorFlow
API Name: {api name}
I need to write a C++ test harness for the TensorFlow C++ frontend operation {api_name}.

API Documentation
{API document}

Your Goal: Generate C++ code for the fuzz.cpp file.
Requirements:
1. Complete the LLVMFuzzerTestOneInput function
2. Tensor Creation:
   1. Parse data type
   2. Determine the rank
   3. Determine the shape
   4. Create input tensor(s) and args for the tf.raw_ops.Acos operation
3. Operation Application: Apply the tf.raw_ops.Acos operation to the created tensors
4. Complete Implementation: The generated code should be complete content for fuzz.cpp

File Structure
Here is the skeleton of the file and helper functions:
```cpp
{test harness template}
```
```cpp
{helper_functions}
```

Important Notes:
Strictly follow the tensor definition in the documentation.

Test harness generated by LLM

```cpp
// ... header files and helper functions ...
tensorflow::DataType parseDataType(uint8_t selector) { ... }
uint8_t parseRank(uint8_t byte) { ... }
std::vector<int64_t> parseShape(const uint8_t* data, size_t& offset,
size_t total_size, uint8_t rank) { ... }
template <typename T>
void fillTensorWithData(tensorflow::Tensor& tensor,
      const uint8_t* data, size_t& offset, size_t total_size) { ... }

extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data,size_t size) {
  tensorflow::Scope root = tensorflow::Scope::NewRootScope();
  try {
    tensorflow::DataType dtype = parseDataType(data[offset++]); ❶
    uint8_t rank = parseRank(data[offset++]); ❷
    std::vector<int64_t> shape = parseShape(data, offset, size, rank); ❸
    tensorflow::Tensor input_tensor(dtype, tensor_shape);
    fillTensorWithDataByType(input_tensor, dtype, data, offset, size); ❹
    auto input_placeholder = tensorflow::ops::Placeholder(root,dtype);
    auto acos_op = tensorflow::ops::Acos(root, input_placeholder); ❺
    tensorflow::ClientSession session(root);
    std::vector<tensorflow::Tensor> outputs;
    tensorflow::Status status = session.Run({{input_placeholder,
                                input_tensor}},{acos_op}, &outputs);
    // error handling and cpu-gpu oracle are omitted for brevity ❻❼
  } catch (const std::exception& e) {
    tf_fuzzer_utils::logError(
            "Execution error: " + std::string(e.what()), data, size);
    return -1;
  }
  return 0;
}
```

Fig. 4: Prompt for generating a test harness for the API `tf.raw_ops.Acos` and corresponding harness.

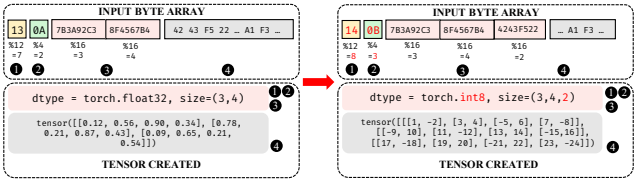


Fig. 5: Input representation in test harness. Byte values in the input array are shown in hexadecimal.

tensor would change. A more dramatic change occurs if the fuzzer changes the second byte (❷ in the input byte array) from `0x0A` to `0x0B`. In that case, the shape would change to (3, 4, 2), affecting the amount of data to be read from the input byte array. FLASHFUZZ uses a fixed input byte array size of 128 bytes. When the required data exceeds available bytes, missing values default to 1. Recall that CGF fuzzers are capable of identifying which of the generated inputs contribute to coverage and should be retained (§ II-C). We additionally avoid retaining inputs that result in runtime exceptions of the API as those are likely invalid.

IV. METHOD

This section describes the method we use to conduct the study from this paper. Figure 6 shows the workflow we follow to test DL library APIs. First, the *test harness generator* (§ IV-A) produces a test harness for a given API. Second, we start a *fuzzing campaign* (§ IV-B) using the test harness obtained in the previous step. The failures detected in this stage, such as crashes and numerical divergences, can potentially be false alarms. Hence, in the third step, we validate if the failure can be reproduced both through the C++ API and the Python API. We elaborate these steps next.

A. Test Harness Generator

We develop a test harness generation method to support this study. The method is based on three observations: (1) the majority of the functionality of a DL API is in the kernel code, which is implemented in C++ [16] (§ II-B); (2) DL APIs typically manipulate tensor-like data structures; and (3) recent experiments have shown that Large Language Models are capable of synthesizing test harnesses that can parse raw data into structured inputs [35], [36], [23].

Generating semantically-valid inputs to test DL library APIs can be challenging. First, the test harness must be capable of converting array of bytes (i.e., the input of highly-optimized CGFs) into the data structures that an API uses (i.e., tensor-like data structures). Second, the generated inputs should satisfy complex input constraints of an API. To mitigate these problems, FLASHFUZZ prompts an LLM with the API metadata, a C++ harness template, and helper functions, which restricts the space of possible harnesses the LLM can generate.

The leftmost box from Figure 6 illustrates the LLM-based approach that FLASHFUZZ employs to synthesize test harnesses. First, the user provides an API name as input. Second, FLASHFUZZ uses the name of the API, its documentation, helper functions (e.g., the function `parseTensorData` parses a tensor object from a byte array), and a template for the harness to formulate a prompt describing the task for the LLM. Third, FLASHFUZZ uses the prompt to query the LLM. Finally, FLASHFUZZ checks if the obtained harness is plausible by

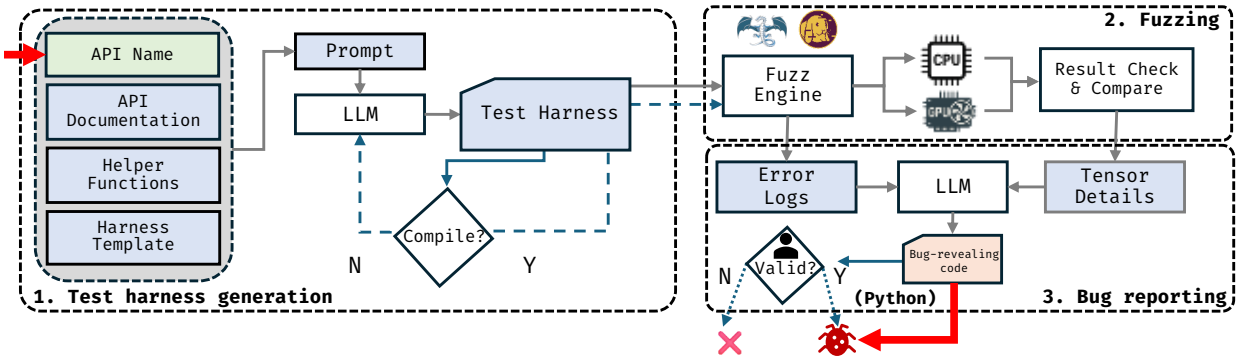


Fig. 6: Workflow of Method Used to Fuzz Deep Learning Library APIs.

(1) checking if it compiles and (2) checking if the target API is the focal method in the harness. If one of these criteria is not met, the prompt is updated with a description of the observed problem. This process repeats until succeeding or reaching a budget on the number of iterations. If successful, FLASHFUZZ reports a harness on output.

API documentation. FLASHFUZZ expects the DL library to provide API documentation through the attribute `__doc__`. For example, FLASHFUZZ retrieves the documentation of the arc cosine API by reading the field `tf.raw_ops.Acos.__doc__`. This documentation (§ III-A) provides valuable information about the input constraints for the API.

Helper functions. Helper functions provide the infrastructure for converting raw binary input data into structured program data. They enable creation of tensors with varying data types, dimensional ranks, shapes, and values. The main helper functions we develop are:

1. `parseDataType`: maps a byte to a scalar data type;
2. `parseRank` extracts the number of dimensions of a tensor;
3. `parseShape` extracts the size of each dimension of a tensor;
4. `createTensor` uses information obtained with other helper functions to create a tensor object.

We provide these methods as a set of utility functions that the LLM can use. It is important to note that the LLM can modify the definition of a helper function. In the arc cosine example from Figure 4, the LLM uses the API documentation to modify the helper function `parseDataType` making it wrap the numeric parameter `select` within the 0-5 range and return a TensorFlow type that is among the options listed in the documentation (e.g., `bfloat16`, `half`, etc.).

Harness template. The harness template defines the skeleton for the test harness that the LLM produces on output. It incorporates necessary header files to compile the harness, namespace declarations, and standardized elements to give structure to the test harness the LLM generates.

1) Transforming Fuzzer-Generated Input Bytes: We observe that the LLM employs three patterns to transform fuzzer-generated input byte arrays for a certain API. Figure 7 illustrates the three patterns on an input for the API `tf.raw_ops.Conv2D`, which has complex constraints [37]. This API requires multiple input arguments including an input tensor (`input`), a filter tensor (`filter`), and a strides vector

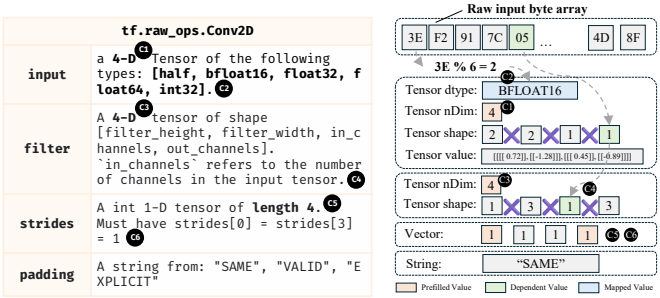


Fig. 7: Illustration of LLM transformation patterns.

(`strides`). The main constraints of the API are: (**c1**) the input tensor must be 4-dimensional, (**c2**) the data type of the input tensor must be one among six options, (**c3**) the filter tensor must be 4-dimensional, (**c4**) the filter tensor must have a shape compatible with the input tensor. (**c5**) the strides vector must contain exactly four elements, (**c6**) both the first and last elements of the strides vector must be 1.

Value Range Mapping. The raw data may contain arbitrary numerical values outside the range admitted by a parameter of an API. The LLM-generated test harness addresses this issue by mapping out-of-range values to in-range values of a numerical API parameter. Figure 7 highlights the usage of the modulo operator to enforce constraint **c2**. The operator maps the first byte in the input array to one of the six choices of data types listed in the documentation of `Conv2D` (corresponding to elements in the `tensor` parameter of the API). We observe this kind of transformation in the helper function `parseDataType`.

Value Prefilling. Certain APIs require specific values as input parameters. The constraints **c1**, **c3**, and **c6** from `Conv2D` illustrate that. The LLM-generated harness automatically identifies these constraints and prefills the necessary values in the input arguments to ensure proper API functionality. For example, Figure 7 shows that the vector `strides` is prefilled with the value one at both ends (constraint **c6**). Likewise, the figure also shows that the attributes `nDim` from the tensors `input` and `filter` are initialized with four dimensions (**c1** and **c3**).

Value Dependence. The LLM-generated harness demonstrates the capability to inherit appropriate values from the surrounding context, ensuring that dependent parameters maintain consistency with their parent operations or previously established

values. In Figure 7, the shape of the `filter` tensor is derived from the channel dimension of the `input` tensor, `in_channels`. More specifically, `Conv2D` requires the size of the last dimension of the `input` tensor (`in_channels`, which equals to 1) to match the size of the third dimension of the `filter` tensor, which is also 1. The `in_channels` represents the number of channels that the convolution processes. The generated harness reads the input tensor's channel count and automatically sets the filter's corresponding dimension, satisfying constraint **c4** instead of using values from the raw data.

B. Fuzzing

FLASHFUZZ uses standard coverage-guided fuzzing (§ II-C) to generate inputs. As in prior work (§ VII), FLASHFUZZ uses differential testing oracles to detect bugs across different devices (i.e., CPU and GPU) during fuzzing. Those oracles observe two kinds of divergences between CPU and GPU executions on the same input: (1) **Output.** FLASHFUZZ flags a potential bug when outputs differ above a threshold of relative tolerance 10^{-2} and absolute tolerance 10^{-3}. This value is consistent with those used in previous deep learning library testing research [15], [5]; (2) **Exception.** FLASHFUZZ flags a potential bug when the execution on a device raises an exception while the other proceeds without errors.

C. Bug Reporting

The preceding steps aim to find inputs that cause a C++ API to exhibit unexpected behavior. Unfortunately, *false alarms* can occur. Despite our efforts to generate test harnesses that avoid invalid inputs, several sources of imprecision remain, including incomplete documentation and the inherent imprecision of LLM-generated harnesses. When available, FLASHFUZZ leverages the input-validity checkers provided by DL libraries' C++ APIs (e.g., PyTorch). These checkers raise runtime errors that FLASHFUZZ can detect by parsing log messages and classifying the run as an "invalid input" execution. Our test-harness handles such errors. However, this design is not uniformly adopted across DL libraries; TensorFlow, for instance, performs some validity checking only in its Python APIs, so some inputs that reach this stage may still be spurious when using FLASHFUZZ with TensorFlow.

To mitigate this issue, FLASHFUZZ reports a bug only if it can be reproduced via the Python APIs. All bugs reported by FLASHFUZZ are manually triaged through a three-step process. First, we reproduce the crash by re-executing the test harness with the crashing input and verifying the error messages and stack traces in the logs. Second, we manually instrument the test harness with additional debug output. Third, we rerun the instrumented harness to collect the necessary diagnostics and attempt to reproduce the bug through the Python API, aided by an LLM chatbot. Finally, we prepare a bug report to submit to developers including the Python script to reproduce the bug and a description to explain the problem.

V. EVALUATION

We answer the following research questions:

RQ1: How does FLASHFUZZ compare with the SoTA on standard evaluation metrics?

RQ2: How effective is FLASHFUZZ in revealing new bugs in the latest versions of popular DL libraries?

RQ3: How important are the components of FLASHFUZZ?

RQ1 evaluates how FLASHFUZZ compares against SoTA techniques on standard metrics from the literature (in API-level fuzzing of DL libraries): coverage, validity ratio, and the number of bugs detected. RQ2 reports on FLASHFUZZ's ability to uncover bugs in the latest versions of PyTorch and TensorFlow. RQ3 reports an ablation study to assess the impact of FLASHFUZZ's design choices.

A. Answering RQ1: How does FLASHFUZZ compare with the SoTA on standard evaluation metrics?

We compare FLASHFUZZ and SoTA techniques using standard evaluation metrics from the literature.

Comparison Baselines. We compare FLASHFUZZ against three recently-proposed fuzzing techniques for finding bugs in deep learning libraries, namely ACETEST [22], PATHFINDER [16], and TITANFUZZ [15]. We select ACETEST and PATHFINDER because they analyze the kernel implementation of the operators in the backend of the libraries, like FLASHFUZZ does and we select TITANFUZZ because it is a well-known representative of a technique that uses Large Language Models (LLMs) to generate inputs (FuzzGPT [38] claims to outperform TITANFUZZ but the tool is not publicly available). TITANFUZZ was initially configured with `facebook/incoder-1B`, which is a relatively outdated model (released April 2022). So, for a fair comparison, we communicated with the authors of TITANFUZZ and updated TITANFUZZ LLM to use `qwen2.5-coder:7b` [39], which is newer (released July 2025) and larger compared to the original model. We validated that TITANFUZZ configured with the Qwen model outperforms the original configuration in terms of coverage, validity ratio, and the number of detected bugs. We reported the results to the authors. To avoid bias, we use the scripts from the artifacts of the comparison baselines to run their experiments and obtain results.

APIs Analyzed. Different techniques support different sets of APIs. Table I shows the breakdown of APIs that FLASHFUZZ currently supports and the number of APIs in common with the comparison baselines. To contextualize the scope of our targets, we quantified how many backend APIs are targeted by FLASHFUZZ and how many are supported. For PyTorch, we programmatically enumerated backend APIs (e.g., `torch.Tensor.*`, `torch.nn.*`, `torch.fft.*`, `torch.linalg.*`, `torch.special.*`, `torch.jit.*`, and storage classes) and obtained a total of 1,576 APIs. Of these, FLASHFUZZ successfully generates a harness for 1,271; the remaining APIs are excluded as FLASHFUZZ could not generate a harness that passes its validation process. For TensorFlow, we enumerated all operators under `tf._api.v2.raw_ops` and obtained a total of 1,452 "raw operators". Of these, FLASHFUZZ successfully generates a harness for 786; the remaining APIs are similarly excluded.

Metrics. We use *coverage*, *validity ratio*, and *bug detection ability* as comparison metrics as they have been previously

TABLE I: Number of APIs FLASHFUZZ supports and number of APIs in common between FLASHFUZZ and each comparison baseline (columns "∩ <baseline-name>").

	FLASHFUZZ	∩ ACETEST	∩ PATHFINDER	∩ TITANFUZZ
PyTorch	1,271	284	529	469
TensorFlow	786	519	478	643

used in the literature to evaluate API-level fuzzers [22], [16], [15]. For *coverage* and *validity ratio*, we run all the techniques on PyTorch 2.2.0 and TensorFlow 2.16.1. For *bug detection ability*, we run the techniques on the latest library version, namely PyTorch 2.7.0 and TensorFlow 2.19.0.

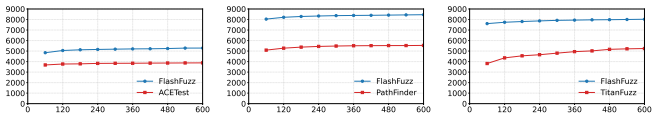
Coverage. As in the PATHFINDER evaluation [16], we measure branch coverage of C++ kernel code. To compute coverage, we instrument the PyTorch and TensorFlow libraries to report branch coverage by using the LLVM flags `−fprofile−instr−generate` and `−fcoverage−mapping` flags. We use llvm-cov [40] to analyze the coverage data and filter coverage information to focus only on kernel functions: `aten/src/ATen/native` for PyTorch and `tensorflow/core/kernels` for TensorFlow. Filtering has been used in the evaluation of prior work [16]; the rationale is to prevent irrelevant, possibly large, portions of code from distorting coverage numbers. To obtain coverage data, we configured the techniques as follows. For ACETEST and TITANFUZZ, we calculate coverage based on their generated files. For PATHFINDER, we modified their artifacts to use `llvm−cov` (instead of `lcov` [41]) to ensure consistency in coverage measurement across all techniques.

Validity ratio. As commonly done in the literature [22], [16], [15] due to the lack of formal API specifications, validity ratio is defined as the percentage of generated inputs that do not trigger input validation errors when executed in the target API.

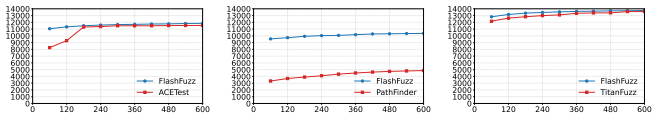
Bug detection ability. To assess the ability of techniques to reveal bugs, we count the number of unique crashes each technique generates, i.e., inputs that cause the program to crash. We define a *unique crash* as a crash that has a distinct stack trace. We manually examine the stack traces and error messages to deduplicate the crashes.

Evaluation setup. For the generation stage of TITANFUZZ, we used two machines equipped with AMD EPYC 7742 CPU and 8 NVIDIA RTX V100 GPUs. For the coverage calculation stage of TITANFUZZ and all the other experiments, we used two CPU-only machines equipped with AMD EPYC 9684X CPU. FLASHFUZZ uses Claude Sonnet 4.0 [42] for test harness generation. We set a budget of 3 iterations per API for testharness generation, with an overall LLM cost of approximately $50 across all APIs. All generaeted harnesses are released in our replication artifact [25] We use a 10-minute fuzzing budget per target API.

1) Results for coverage: Figure 8 presents the progress of branch coverage for each technique over time, with a budget of 10 minutes. We report results for each pair of techniques on PyTorch and TensorFlow. Recall that techniques support different sets of APIs (Table I). To ensure a fair comparison, we only consider the APIs that both techniques support. Each data point represents the merged coverage over all APIs that



(a) FLASHFUZZ vs. {ACETEST,PATHFINDER,TITANFUZZ} on PyTorch.



(b) FLASHFUZZ vs. {ACETEST,PATHFINDER,TITANFUZZ} on TensorFlow.

Fig. 8: Coverage progress for FLASHFUZZ and baselines.

both techniques support. We aggregate coverage across APIs by merging the branch coverage for each API. The improvement percentages are computed as the difference in coverage between FLASHFUZZ and each baseline. Overall, results show that FLASHFUZZ consistently outperforms all baselines in both PyTorch and TensorFlow. Note that the differences between FLASHFUZZ and ACETEST on TensorFlow are marginal after 3 minutes whereas the difference between FLASHFUZZ and TITANFUZZ on TensorFlow is marginal throughout.

2) Results for validity ratio: Table II presents the cumulative results across all APIs supported by a given pair of techniques, showing (1) the total number of inputs each tool generates, (2) the number of valid inputs, and (3) the percentage of valid inputs generated. We observe that the number of valid inputs FLASHFUZZ generates within the time budget is at least 2.5 times higher and at most 1271.3 times higher compared to the baselines, demonstrating FLASHFUZZ's efficiency. See highlighted cells.

In PyTorch, we observe that FLASHFUZZ produces valid inputs at similar ratios compared to the baselines, but the number of valid inputs is significantly higher in FLASHFUZZ. In TensorFlow, the ratio of valid inputs of FLASHFUZZ is 1.7x to 5.4x higher than the baselines. The reason for such large difference is that TensorFlow performs some validity checks only in the Python frontend, not in the C++ backend where FLASHFUZZ operates (§ VI-B).

3) Results for bug detection ability: We compare FLASHFUZZ against each baseline by measuring the number of unique crashes. FLASHFUZZ discovers substantially more unique crashes than all baselines. On PyTorch, it uncovers 5 crashes compared to ACETEST (0), PATHFINDER (0), and 17 crashes compared to TITANFUZZ (1), with no overlaps between FLASHFUZZ and any baseline. On TensorFlow, it uncovers 21 crashes vs. ACETEST (3), 16 vs. PATHFINDER (5), and 25 vs. TITANFUZZ (13), with overlaps of 2, 4, and 1 crashes, respectively.

Summary RQ1: FLASHFUZZ outperforms all techniques on all metrics. Most importantly, the ability to detect bugs is significantly higher in FLASHFUZZ compared to the baselines.

B. Answering RQ2: How effective is FLASHFUZZ *in revealing new bugs in the latest versions of popular DL libraries?*

Table III shows the status of the bugs we reported to developers when fuzzing APIs "in the wild". More precisely,

TABLE II: Total number of inputs each tool generates and their valid input rates.

PyTorch						
Tool	**ACETEST**	**FLASHFUZZ**	**PATHFINDER**	**FLASHFUZZ**	**TITANFUZZ**	**FLASHFUZZ**
Total	9,267.4K	158,027.2K (17.1×)	58,333.2K	270,083.6K (4.6×)	206.8K	244,433.4K (1182.0×)
Valid	4,918.4K	104,971.4K (21.3×)	36,126.7K	170,117.1K (4.7×)	126.6K	160,943.9K (1271.3×)
Ratio (%)	53.07	66.43 (1.3×)	61.93	62.99 (1.0×)	61.22	65.84 (1.1×)
TensorFlow						
Tool	**ACETEST**	**FLASHFUZZ**	**PATHFINDER**	**FLASHFUZZ**	**TITANFUZZ**	**FLASHFUZZ**
Total	38,451.8K	39,047.2K (1.0×)	20,392.2K	34,662.0K (1.7×)	110.7K	59,252.4K (535.3×)
Valid	14,843.5K	37,183.2K (2.5×)	3,605.1K	32,866.4K (9.1×)	60.0K	54,549.7K (909.2×)
Ratio (%)	38.60	95.23 (2.5×)	17.68	94.82 (5.4×)	54.23	92.06 (1.7×)

TABLE III: Bug statistics.

(a) Status of bugs reported to developers.

Framework	Reported	Duplicate	Confirmed	Fixed	Pending
PyTorch	32	0	31	14	1
TensorFlow	22	0	22	1	0

(b) Types of bugs (Confirmed + Pending).

Framework	Aborted	Segfault	FPE	Memory Overflow	Internal Exception	Inconsistency (CPU-GPU)	Σ
PyTorch	2	6	6	1	3	14	32
TensorFlow	8	7	2	0	2	3	22
Σ	10	13	8	1	5	17	54

we run each API for 8 hours. The results show that developers confirmed the majority of the bugs (˜98%) we reported, suggesting that our mechanism for filtering false alarms was effective. Table IIIb breaks down the confirmed bugs into six categories. We use a categorization similar to one used in recent prior work (e.g., PATHFINDER [16]). These results suggest that FLASHFUZZ is capable of revealing a diverse range of bugs.

Summary RQ2: FLASHFUZZ was highly effective in finding bugs, revealing 54 previously-unknown bugs; 32 in PyTorch and 22 in TensorFlow.

C. Answering RQ3: How important are the components of FLASHFUZZ?

To assess the contribution of each component of FLASHFUZZ, we performed an ablation study in which we removed, one at a time, two elements from the LLM prompt: (i) API documentation and (ii) helper functions. We did not ablate the template component because it is required to compile harnesses, measure validity, and handle errors. We evaluated the effect on three metrics: validity ratio, branch coverage, and the number of crashes found. Results are summarized in Table IV. For each framework (PyTorch and TensorFlow), we constructed the target set by combining APIs on which FLASHFUZZ previously revealed crashes with a random sample from the supported API list, yielding 100 targets per framework. We allocated a 10-minute budget per API. In Table IV, arrows indicate whether a metric increased (↑) or decreased (↓) relative to the full FLASHFUZZ configuration.

The results show that documentation and helper functions affect the two frameworks differently. For TensorFlow, both components substantially aid validity and coverage: removing documentation reduces validity from 92.89% to 61.45% and coverage from 1336 to 593 (crashes remain 9); removing helper functions also reduces validity (to 76.00%) and coverage (to 581) and eliminates crashes ($9 \rightarrow 0$). Removing both yields the lowest validity (53.55%) and coverage (356), with a single crash. For PyTorch, documentation improves both validity and coverage: without documentation, validity drops from 64.75% to 34.44% and coverage from 2501 to 1985 (crashes remain 2). Removing helper functions increases validity (to 77.04%) but substantially reduces coverage (to 1140) and eliminates crashes ($2 \rightarrow 0$). Overall, documentation consistently boosts validity and coverage; helper functions trade higher coverage and crash discovery for lower validity in PyTorch, but aid both validity and coverage in TensorFlow.

Summary RQ3: Overall, results demonstrate that the use of documentation and helper functions are essential for the performance we obtain using libFuzzer with the test harnesses FLASHFUZZ generates.

VI. DISCUSSION

We discuss a small sample of bugs we find (Section VI-A), known limitations of this study (Section VI-B), threats to validity (Section VI-C), and lessons learned (Section VI-D).

A. Bugs detected by FLASHFUZZ

We describe two bugs that FLASHFUZZ detected; one in PyTorch and one in TensorFlow. The full list of bugs is available in our artifacts repository.

PyTorch Bug #153337. PyTorch's `torch.combinations` API computes r-length combinations of elements from a 1D tensor. FLASHFUZZ detects a bug in this API, as shown by the code snippet below that contains a fault-revealing input tensor (11 complex numbers) and an API call that requests all length-10 combinations with replacement. This input is valid according to the API documentation, but it causes the execution to consume excessive memory and hang.

```
tensor = torch.tensor([-1.7267e-23 + 5.2018e-31j, ...,
    -1.2775e+17 - 5.5035e-17j], dtype=torch.cfloat,)
torch.combinations(tensor, r=10, with_replacement=True)
```

Listing 3: Performance bug #153337 in PyTorch 2.7.0

Root Cause: The C++ backend implementation of this API (`aten/src/ATen/native/Itertools.cpp`) builds the entire r-fold Cartesian product using `at::meshgrid` and only then filters it. As a result, this API allocates large intermediate tensors of size n^r (data and indices), ignoring the fact that

TABLE IV: Ablation study results for FLASHFUZZ. For each metric, the right-adjacent column (Δ) shows the change relative to the full FLASHFUZZ configuration (FlashFuzz); arrows indicate increases (↑) or decreases (↓), and ↔ denotes no change.

Configuration	PyTorch						TensorFlow					
	Validity	Δ	Coverage	Δ	Crashes	Δ	Validity	Δ	Coverage	Δ	Crashes	Δ
FlashFuzz	64.75	—	2501	—	2	—	92.89	—	1336	—	9	—
w/o Documentation	34.44	30.31% ↓	1985	516 ↓	2	0 ↔	61.45	31.44% ↓	593	743 ↓	9	0 ↔
w/o Helper Functions	77.04	12.29% ↑	1140	1361 ↓	0	2 ↓	76.00	16.89% ↓	581	755 ↓	0	9 ↓
w/o Doc+Helper Func	23.70	41.05% ↓	420	2081 ↓	0	2 ↓	53.55	39.34% ↓	356	980 ↓	1	8 ↓

the final output will only be of size $\binom{n}{r}$ or $\binom{n+r-1}{r}$. Since n^r grows extremely fast, the call consumes a lot of memory and hangs. Note that this is a deep C++ backend design bug; not a Python API misuse. Following our bug report, other users have also reported similar issues in the same API, showing that this is a practical and common problem.

TensorFlow Bug #94117. The TensorFlow API `tf.raw_ops.ResourceSparseApplyProximalAdagrad` performs sparse updates on entries in the input tensor variables according to the FOBOS [43] algorithm.

```
large_val = 5.24393461e+36
var = tf.Variable([[large_val]*3, [large_val]*3], dtype=tf.
    float32, name="var")
accum = tf.Variable([[large_val]*3, [large_val]*3], dtype=
    tf.float32, name="accum")
lr = l1 = l2 = tf.constant(large_val, dtype=tf.float32)
grad = tf.constant([7.90505e+31], dtype=tf.float32)
indices = tf.constant([0], dtype=tf.int32)
tf.raw_ops.ResourceSparseApplyProximalAdagrad(var=var.
    handle, accum=accum.handle, lr=lr, l1=l1, l2=l2, grad=
    grad, indices=indices, use_locking=False)
```

Listing 4: Crash bug #94117 present in TensorFlow 2.19.0

However, when the input tensors `var` and `accum` contain large values (in the order of 10^{30}), they cause the API to crash with a fatal error: `Checkfailed: d < dims()(1vs.1)`. Ideally, the API should handle large input values gracefully without crashing. Developers confirm the issue.

B. Limitations

Accuracy of harnesses in generating valid inputs. The ability of our method to produce valid inputs depends largely on (1) how accurately the LLM interprets the API documentation to identify constraints, and (2) how accurately it transforms raw bytes into valid inputs. We observe that LLMs capture these requirements for a majority of harnesses, but we still see failures. For PyTorch, we enumerated 1,576 C++ candidate APIs: FLASHFUZZ built 1,271 (81%) runnable harnesses that execute the API under test, while 227 (14%) failed to build due to compilation/linking errors, 65 (4%) failed due to missing target API, and 13 (0.8%) failed at runtime. For TensorFlow, among 1,452 candidate APIs, FLASHFUZZ built 786 (54%) runnable harnesses that execute the API under test, while 635 (44%) failed due to compilation/linking errors, 25 (2%) failed due to missing target API, and 6 (0.4%) failed at runtime.

We examined several failing harnesses. Out of ten randomly sampled harnesses that failed to build or run, four failed to build because the target API was missing in the backend, and six failed due to syntax errors or incorrect API usage. In a separate random sample of ten missing target API cases, all were caused by the LLM not following instructions and selecting a similar API instead. Four of the runtime failures were silent, which are difficult for libFuzzer to detect.

Fuzzing on Backend vs FrontEnd. Since FLASHFUZZ focuses on fuzzing the C++ backend, it may miss bugs that are present only in the Python frontend or the glue code between the frontend and backend. However, in DL frameworks, the lion share of the code typically resides in the backend. Further, some frameworks like TensorFlow have additional validity checks in the frontend that are not present in the backend. As a result, we observe that some potential bugs found by FLASHFUZZ turn out to be false positives when tested with the Python frontend.

Performance Considerations. We configured the fuzzer for maximum throughput. We capped maximum input length at 128 bytes, limited mutation depth to 8, enabled two threads to overlap I/O on a single core, and used a single in-process runner to avoid TensorFlow/PyTorch initialization costs. These configurations are important to maximize throughput, but we did not explore all possible optimizations. Further, we did not use any domain-specific mutators, which may further improve coverage and bug-finding ability.

C. Threats to Validity

External Validity. A threat to the external validity of FLASHFUZZ concerns its generalizability across different deep learning libraries. This threat arises because our evaluation focuses primarily on PyTorch and TensorFlow, which may limit the applicability of our findings to other frameworks. However, PyTorch and TensorFlow are still the most widely used and most general deep learning libraries, and so our findings may still be relevant to other emerging libraries that share similar design principles, such as JAX [44].

Internal Validity. An internal threat to validity is the reliability of our LLM-based test harness generation. The quality of generated harnesses depends on the LLM's ability to correctly interpret API documentation and produce correct C++ test harness code. To mitigate this threat, we implemented a multi-stage validation process for the generated harnesses. This process includes compilation checks, manual inspection of a sample of harnesses, and procedures to ensure that detected crashes originate from the target API rather than the harness code itself. Furthermore, FLASHFUZZ is not tied to a specific LLM. While our main evaluation uses Claude Sonnet 4.0, we verified that FLASHFUZZ also produces valid harnesses with Gemini 3 Pro [45] and GLM 5 [46], demonstrating that the approach is not limited to a single LLM.

D. Lessons Learned and Future Directions

We list lessons learned and directions for future work:

Coverage-guided fuzzing should be used in future evaluations of techniques for API-level fuzzing of DL libraries. Our results provided initial yet strong evidence that coverage-guided fuzzing is effective at finding bugs in DL APIs. In contrast to prior findings that coverage guidance is inefficient due to a lack of necessary structural awareness [14], we show that FLASHFUZZ's integration with libFuzzer is highly efficient. Further, our results show that with systematic guidance, LLMs can generate correct harnesses even for complex domains like DL libraries. As a result, decades of research in coverage-guided fuzzers can be brought to bear instead of developing fuzzers from scratch. Hence, we recommend the future work to use coverage-guided fuzzing as a (strong) baseline.

There is room to improve harness generation for DL APIs. While the LLM-based approach of FLASHFUZZ can generate harnesses for a large number of APIs (see Table I), there are still many APIs (305 for PyTorch, 666 for TensorFlow) we could not generate harnesses for. Furthermore, even when we generate harnesses, the coverage and validity ratio is low for some APIs. Recent LLM-based techniques (e.g., CKGFuzzer [24]) that uses knowledge graphs to generate fuzz harnesses can be combined with FLASHFUZZ to further improve harness generation. We leave it for future work.

Triaging needs more attention. We spent 10-30 minutes triaging each potential bug we found. After finding a likely bug, we needed to check if it was unique by analyzing stack traces and then translate C++ inputs (byte arrays) into Python API calls (§ IV-C). We use an LLM for this translation, but manual intervention is still required, e.g., when tensor data exceeds the LLM context or the generated Python code is incorrect. Furthermore, inputs sometimes require minimization before reporting. To sum up, triaging failures is a very important part of the fuzzing process. The community should give credit to research on bug triage as it would greatly benefit the ability of fuzzers to find real bugs.

VII. RELATED WORK

Test Harness Generation. Recent approaches incorporate auxiliary data for harness generation but face scalability or structural limitations. CKGFuzzer [24] and ElFuzz [47] utilize code knowledge graphs and LLMs, respectively, but often lack explicit API structure awareness. Nexzzer [48] extracts API relations to generate sequences, whereas FLASHFUZZ focuses primarily on individual API properties. PromptFuzzer [23] mutates LLM prompts for coverage but is not designed for DL-specific structures like tensors. In contrast, FLASHFUZZ explicitly parses tensor representations and complex constraints for DL library APIs.

Oracles. Due to the lack of formal specifications, prior work relies on metamorphic oracles (e.g., checking properties under transformations [49], [50]) and differential oracles. CRADLE [7] compares CPU/GPU backends, while Li et al. [51] use LLMs for cross-library testing. We focus on differential oracles to detect inconsistencies and crashes.

Model-level Fuzzing. Model-level fuzzers generate full models to test libraries. AUDEE [8], LEMON [9], and MUFFIN [10] mutate model weights, architectures, or computation graphs. Others target compilers: NNSmith [11] uses SMT solvers, and NeuRI [12] infers operator rules. However, model-level approaches often struggle with input diversity due to strict inter-API constraints, motivating API-level fuzzing.

API-level Fuzzing. API-level fuzzers target either the Python frontend or the C++ backend. *Frontend:* Since Python lacks explicit types, tools like FreeFuzz [5] and DocTer [4] mine code snippets or documentation to infer argument constraints. DeepREL [13] leverages API signature similarities to maximize coverage. LLM-based approaches, including TITANFUZZ [15] and FuzzGPT [38], use LLMs to directly generate and mutate test inputs (i.e., Python programs), keeping the LLM in the fuzzing loop. In contrast, FLASHFUZZ uses an LLM only once to synthesize C++ test harnesses, similar to how MetaMut [52] and IssueMut [53] use LLMs to synthesize compiler mutators; fuzzing is then driven entirely by a traditional coverage-guided fuzzer (libFuzzer). While TITANFUZZ and FuzzGPT target the Python frontend, FLASHFUZZ targets the C++ backend, enabling lightweight, high-throughput coverage-guided fuzzing. *Backend:* Backend fuzzing benefits from the strong typing of C++. IvySyn [14] uses type-aware mutation but operates as a black box. Gray-box methods include PATHFINDER [16], which uses inductive synthesis for path constraints, and ACETest [22], which traces error-handling code. Unlike ACETest and PathFinder, FLASHFUZZ uses coverage guidance and test harnesses that can generate valid inputs from API constraints. Concurrent work, Centaur [54], uses neurosymbolic constraint learning to test DL libraries. BugsInDLLs [55] provides a database of reproducible DL library bugs.

VIII. CONCLUSIONS

This paper reports the first in-depth study on using coverage-guided fuzzing (CGF) to test deep learning (DL) library APIs. CGF has been very successful to find bugs in various domains. Yet, it remains largely unexplored to find bugs in DL library APIs. CGF requires test harnesses to drive fuzzing campaigns. As such, to support this study, we develop a tool that wraps the coverage-guided fuzzer libFuzzer [18] with the capability of generating harnesses for DL APIs. We call that tool FLASHFUZZ. Results show that FLASHFUZZ outperforms three SoTA recently-proposed techniques (namely, ACETEST, TITANFUZZ, and PATHFINDER) in terms of coverage, validity rate, and ability to reveal crashes. A large-scale study demonstrates the practical utility of FLASHFUZZ, revealing **54** previously unknown bugs in two recent versions of PyTorch and TensorFlow. The main lesson we learned from this study is that CGF proves competitive with specialized API-level DL library fuzzers. We recommend future work adopt it as a strong baseline.

Acknowledgments. We thank the reviewers for their comments. This work was partially supported by the U.S. National Science Foundation grant No. CCF-2349961. We thank Google for the Google Cloud Platform credits.

REFERENCES

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: a system for large-scale machine learning," in *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, 2016, pp. 265–283.

[2] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[3] L. Jia, H. Zhong, X. Wang, L. Huang, and X. Lu, "The symptoms, causes, and repairs of bugs inside a deep learning library," *Journal of Systems and Software*, vol. 177, p. 110935, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121221000327

[4] D. Xie, Y. Li, M. Kim, H. V. Pham, L. Tan, X. Zhang, and M. Godfrey, "Docter: Documentation-guided fuzzing for testing deep learning api functions," in *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, Jul. 2022. [Online]. Available: https://www.cs.purdue.edu/homes/lintan/papers.html

[5] A. Wei, Y. Deng, C. Yang, and L. Zhang, "Free lunch for testing: Fuzzing deep-learning libraries from open source," in *Proceedings - International Conference on Software Engineering*, vol. 2022-May, 2022.

[6] J. Chen, Y. Liang, Q. Shen, J. Jiang, and S. Li, "Toward understanding deep learning framework bugs," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 6, sep 2023. [Online]. Available: https://doi.org/10.1145/3587155

[7] H. V. Pham, T. Lutellier, W. Qi, and L. Tan, "Cradle: Cross-backend validation to detect and localize bugs in deep learning libraries," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 1027–1038.

[8] Q. Guo, X. Xie, Y. Li, X. Zhang, Y. Liu, X. Li, and C. Shen, "Audee: automated testing for deep learning frameworks," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '20. New York, NY, USA: Association for Computing Machinery, 2021, p. 486–498. [Online]. Available: https://doi.org/10.1145/3324884.3416571

[9] Z. Wang, M. Yan, J. Chen, S. Liu, and D. Zhang, "Deep learning library testing via effective model generation," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 788–799. [Online]. Available: https://doi.org/10.1145/3368089.3409761

[10] J. Gu, X. Luo, Y. Zhou, and X. Wang, "Muffin: testing deep learning libraries via neural architecture fuzzing," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1418–1430. [Online]. Available: https://doi.org/10.1145/3510003.3510092

[11] J. Liu, J. Lin, F. Ruffy, C. Tan, J. Li, A. Panda, and L. Zhang, "Nnsmith: Generating diverse and valid test cases for deep learning compilers," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS '23. ACM, Jan. 2023. [Online]. Available: http://dx.doi.org/10.1145/3575693.3575707

[12] J. Liu, J. Peng, Y. Wang, and L. Zhang, "Neuri: Diversifying dnn generation via inductive rule inference," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 657–669. [Online]. Available: https://doi.org/10.1145/3611643.3616337

[13] Y. Deng, C. Yang, A. Wei, and L. Zhang, "Fuzzing deep-learning libraries via automated relational api inference," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 44–56. [Online]. Available: https://doi.org/10.1145/3540250.3549085

[14] N. Christou, D. Jin, V. Atlidakis, B. Ray, and V. P. Kemerlis, "IvySyn: Automated vulnerability discovery in deep learning frameworks," in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 2383–2400. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/christou

[15] 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 *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 423–435. [Online]. Available: https://doi.org/10.1145/3597926.3598067

[16] 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 *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2025, pp. 739–739. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICSE55347.2025.00202

[17] F. IO, "Afl++ web site," https://aflplus.plus/, 2025.

[18] K. Serebryany, "Libfuzzer: A library for coverage-guided fuzz testing," https://llvm.org/docs/LibFuzzer.html, 2015.

[19] D. Wang, Z. Zhang, H. Zhang, Z. Qian, S. V. Krishnamurthy, and N. Abu-Ghazaleh, "Syzvegas: Beating kernel fuzzing odds with reinforcement learning," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 2741–2758. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/wang-daimeng

[20] M. Wu, M. Lu, H. Cui, J. Chen, Y. Zhang, and L. Zhang, "Jitfuzz: Coverage-guided fuzzing for jvm just-in-time compilers," in *Proceedings of the 45th International Conference on Software Engineering*, ser. ICSE '23. IEEE Press, 2023, p. 56–68. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00017

[21] J. Pereyda and boofuzz contributors, "boofuzz: Network protocol fuzzing for humans," https://github.com/jtpereyda/boofuzz, 2025, accessed: 2025-09-04.

[22] 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 *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '23. ACM, Jul. 2023, p. 690–702. [Online]. Available: http://dx.doi.org/10.1145/3597926.3598088

[23] Y. Lyu, Y. Xie, P. Chen, and H. Chen, "Promptfuzz: Prompt fuzzing for fuzz driver generation," in *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24)*. ACM, 2024. [Online]. Available: https://doi.org/10.1145/3658644.3670396

[24] H. Xu, W. Ma, T. Zhou, Y. Zhao, K. Chen, Q. Hu, Y. Liu, and H. Wang, "Ckgfuzzer: Llm-based fuzz driver generation enhanced by code knowledge graph," in *2025 IEEE/ACM 47th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE Press, 2025, pp. 243–254. [Online]. Available: https://doi.org/10.1109/ICSE-Companion66252.2025.00079

[25] F. Qin, M. M. A. Naziri, H. Ai, S. Dutta, and M. d'Amorim, "FlashFuzz replication package," https://github.com/ncsu-swat/FlashFuzz, 2025.

[26] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA: MIT Press, 2016, chapter 2.3 on Tensors.

[27] T. G. Kolda, D. Hong, and P. Jain, "Tensor methods in deep learning," *Neurocomputing*, vol. 469, pp. 1–13, 2022.

[28] Y. Bengio, A. Courville, and P. Vincent, "Representation learning: A review and new perspectives," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 35, no. 8, pp. 1798–1828, 2013.

[29] M. Zalewski, "American fuzzy lop," https://lcamtuf.coredump.cx/afl/, 2014.

[30] R. Swiecki, "Honggfuzz: Feedback-driven fuzzer," https://github.com/google/honggfuzz, 2017.

[31] T. Developers, "Tensorflow documentation: tf.math.acos," https://www.tensorflow.org/api_docs/python/tf/math/acos, 2025.

[32] Anonymous, "Fpe crash in torch.fmod with large negative integer modulo value," https://github.com/pytorch/pytorch/issues/153203, 2025.

[33] ——, "Fix fpe crash in torch.fmod," https://github.com/pytorch/pytorch/pull/165833, 2025.

[34] P. Foundation, "Pytorch documentation: torch.fmod," 2025. [Online]. Available: https://docs.pytorch.org/docs/stable/generated/torch.fmod.html

[35] D. Liu, J. Metzman, O. Chang, and G. O. S. S. Team, "Ai-powered fuzzing: Breaking the bug hunting barrier," https://security.googleblog.com/2023/08/ai-powered-fuzzing-breaking-bug-hunting.html, Google, 2023.

[36] V. Vikram, C. Lemieux, J. Sunshine, and R. Padhye, "Can large language models write good property-based tests?" 2024. [Online]. Available: https://arxiv.org/abs/2307.04346

[37] T. Developers, "Tensorflow documentation: tf.raw˙ops.conv2d," https://www.tensorflow.org/api_docs/python/tf/raw_ops/Conv2D, 2025.

[38] 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 *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–13.

[39] B. Hui, J. Yang, Z. Cui, J. Yang, D. Liu, L. Zhang, T. Liu, J. Zhang, B. Yu, K. Lu, K. Dang, Y. Fan, Y. Zhang, A. Yang, R. Men, F. Huang, B. Zheng, Y. Miao, S. Quan, Y. Feng, X. Ren, X. Ren, J. Zhou, and J. Lin, "Qwen2.5-coder technical report," 2024. [Online]. Available: https://arxiv.org/abs/2409.12186

[40] LLVM Project, "LLVM coverage mapping format," https://llvm.org/docs/CommandGuide/llvm-cov.html, 2024.

[41] Linux Test Project, "LCOV coverage mapping format for gcc and clang," https://github.com/linux-test-project/lcov, 2024.

[42] Anthropic, "Introducing claude 4: Claude opus 4 and claude sonnet 4," https://www.anthropic.com/news/claude-4, May 2025, accessed: 2025-09-10.

[43] J. Duchi and Y. Singer, "Efficient online and batch learning using forward backward splitting," *Journal of Machine Learning Research*, vol. 10, pp. 2899–2934, 2009. [Online]. Available: https://www.jmlr.org/papers/volume10/duchi09a/duchi09a.pdf

[44] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang, "JAX: composable transformations of Python+NumPy programs," 2018. [Online]. Available: http://github.com/jax-ml/jax

[45] Google DeepMind, "Gemini 3: Introducing the latest gemini ai model from google," https://blog.google/products/gemini/gemini-3/, Nov. 2025, accessed: 2026-03-31.

[46] GLM-5-Team *et al.*, "Glm-5: from vibe coding to agentic engineering," 2026. [Online]. Available: https://arxiv.org/abs/2602.15763

[47] C. Chen, B. Dolan-Gavitt, and Z. Lin, *ELFUZZ: Efficient Input Generation via LLM-driven Synthesis Over Fuzzer Space*. USA: USENIX Association, 2025.

[48] J. Lin, Q. Zhang, J. Li, C. Sun, H. Zhou, C. Luo, and C. Qian, "Automatic library fuzzing through api relation evolvement," in *Proceedings of the 2025 Network and Distributed System Security Symposium (NDSS 2025)*, 2025.

[49] J. Ding, X. Kang, and X.-H. Hu, "Validating a deep learning framework by metamorphic testing," in *2017 IEEE/ACM 2nd International Workshop on Metamorphic Testing (MET)*. IEEE, 2017, pp. 28–34.

[50] D. Xiao, Z. Liu, Y. Yuan, Q. Pang, and S. Wang, "Metamorphic testing of deep learning compilers," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 6, no. 1, pp. 1–28, 2022.

[51] 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 Trans. Softw. Eng. Methodol.*, vol. 35, no. 4, 2026. [Online]. Available: https://doi.org/10.1145/3735637

[52] X. Ou, C. Li, Y. Jiang, and C. Xu, "The mutators reloaded: Fuzzing compilers with large language model generated mutation operators," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, ser. ASPLOS '24. New York, NY, USA: Association for Computing Machinery, 2025, p. 298–312. [Online]. Available: https://doi.org/10.1145/3622781.3674171

[53] L. Liu, F. Qin, O. Legunsen, and M. d'Amorim, "Learning compiler fuzzing mutators from historical bugs," in *Proceedings of the 23rd IEEE/ACM International Conference on Mining Software Repositories*, ser. MSR '26. IEEE/ACM, 2026. [Online]. Available: https://2026.msrconf.org/details/msr-2026-technical-papers/41/Learning-Compiler-Fuzzing-Mutators-from-Historical-Bugs

[54] M. M. A. Naziri, S. Kim, F. Qin, S. Dutta, and M. d'Amorim, "Testing deep learning libraries via neurosymbolic constraint learning," in *Proceedings of the 48th IEEE/ACM International Conference on Software Engineering*, ser. ICSE '26. IEEE/ACM, 2026. [Online]. Available: https://conf.researchr.org/details/icse-2026/icse-2026-research-track/272/Testing-Deep-Learning-Libraries-via-Neurosymbolic-Constraint-Learning

[55] M. M. A. Naziri, A. K. Singh, B. Wu, F. A. Qin, S. Dutta, and M. d'Amorim, "Bugsindlls : A database of reproducible bugs in deep learning libraries to enable systematic evaluation of testing techniques," in *Proceedings of the 34th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA Companion '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 61–65. [Online]. Available: https://doi.org/10.1145/3713081.3731739