Lightweight Concolic Testing via Path-Condition Synthesis for Deep Learning Libraries

Sehoon Kim, Yonghyeon Kim, Dahyeon Park, Yuseok Jeon*, Jooyong Yi, Mijung Kim†
UNIST, South Korea
{sehoon, yonghyeon, tryness, jsjeon, jooyong, mijungk}@unist.ac.kr
**Korea University, South Korea*
ys_ jeon@korea.ac.kr

***Abstract*—Many techniques have been recently developed for testing deep learning (DL) libraries. Although these techniques have effectively improved API and code coverage and detected unknown bugs, they rely on blackbox fuzzing for input generation. Concolic testing (also known as dynamic symbolic execution) can be more effective in exploring diverse execution paths, but applying it to DL libraries is extremely challenging due to their inherent complexity. In this paper, we introduce the first concolic testing technique for DL libraries. Our technique offers a lightweight approach that significantly reduces the heavy overhead associated with traditional concolic testing. While symbolic execution maintains symbolic expressions for every variable with non-concrete values to build a path condition, our technique computes approximate path conditions by inferring branch conditions via inductive program synthesis. Despite potential imprecision from approximation, our method's light overhead allows for effective exploration of diverse execution paths within the complex implementations of DL libraries. We have implemented our tool, PATHFINDER, and evaluated it on PyTorch and TensorFlow. Our results show that PATHFINDER outperforms existing API-level DL library fuzzers by achieving 67% more branch coverage on average; up to 63% higher than TitanFuzz and 120% higher than FreeFuzz. PATHFINDER is also effective in bug detection, uncovering 61 crash bugs, 59 of which were confirmed by developers as previously unknown, with 32 already fixed.**

***Index Terms*—Fuzzing, Concolic Testing, Deep Learning Libraries**

I. INTRODUCTION

Deep Learning (DL) continues to be a dominant approach in the field of artificial intelligence (AI) and has been extensively used for building various AI systems. It has become a common practice to build AI systems using DL libraries such as PyTorch [1] and TensorFlow [2]. That is, bugs contained in DL libraries can be propagated to the applications and hurt the performance of the implemented models [3], [4], [5], [6], [7]. Accordingly, research on DL library testing has recently been gaining traction.

To test DL libraries, researchers have developed various API-level fuzzing techniques to generate inputs for DL library APIs. These techniques use valid seed inputs obtained from API usage examples [8] or large language models [9], [10], extract API input constraints [11], [12], and borrow inputs from equivalent APIs within the same library [13] or counterpart APIs across different libraries [14]. Although these techniques have proven effective in detecting unknown bugs [11], [8], [13], [9], [10], [12], [14], improving API coverage [13], [9], [10], and enhancing code coverage [12], [14], [9], [10], [8], they still suffer from limited code coverage [15]. This limitation arises because most existing techniques perform blackbox fuzzing, which does not consider the internal code structures of the program under test during input generation. Consequently, low code coverage may affect the bug detection capability of a fuzzing technique, as bugs cannot be detected without being executed.

A potential approach to address low code coverage issues can be greybox fuzzing (such as libfuzzer [16]), which leverages coverage-feedback to guide the input generation to explore untested parts of the code. If a generated input increases coverage, it is added to the seed corpus as a mutation candidate for further fuzzing. However, greybox fuzzing may still suffer from low coverage. Due to the random nature of mutation operators, mutated inputs are highly likely to be rejected by input validity checks and thus often fail to increase code coverage.

Another possible approach to address low coverage can be concolic testing (also known as dynamic symbolic execution or whitebox fuzzing), which systematically explores different execution paths [17], [18], [19], [20]. Concolic testing can be a better solution than greybox fuzzing for improving the coverage since it maintains path conditions and generates inputs that traverse the corresponding paths via constraint solving. Hence, it does not suffer as much as greybox fuzzing from repeated precondition violations. However, performing concolic testing on DL libraries still presents significant challenges due to its well-known drawbacks [21], [22]. First, due to the high software complexity of DL libraries, concolic testing requires *high computational overhead* to handle the symbolic execution and constraint solving for a large number of paths. Second, concolic testing has limited support for programs involving *nonlinear expressions* in the program (e.g., `x * y > w`) due to the difficulty in solving nonlinear constraints. Note that DL libraries such as PyTorch and TensorFlow heavily use nonlinear expressions in their implementations.

To fill this gap, we develop a *lightweight* concolic testing

*The work was done when Yuseok Jeon was at UNIST.
†Corresponding author.

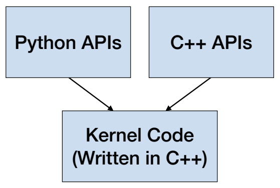


Fig. 1: A layered structure of a typical DL library

```python
def torch.nn.functional.affine_grid(
  theta, size, align_corners=None):

  ...
  # check that shapes and sizes match
  if len(size) == 4:
     if (theta.dim() != 3
        or theta.shape[-2] != 2
        or theta.shape[-1] != 3):
           raise ValueError(some_error_msg)
     spatial_size = size[-2:]
  elif len(size) == 5:
```

(a) Python API

```cpp
inline Tensor torch::nn::functional::affine_grid(
  const Tensor &theta, const IntArrayRef &size,
  bool align_corners = false) {
   ...
   // check that shapes and sizes match
   if (size.size() == 4) {
     TORCH_CHECK(theta.dim() == 3 &&
        theta.size(-2) == 2 &&
        theta.size(-1) == 3,
        some_error_msg);

   } else if (size.size() == 5) {
```

(b) C++ API

Fig. 2: PyTorch's `affine_grid` API in Python and C++

technique designed to systematically *explore diverse execution paths* for DL libraries. The key idea of our approach is derived from how to deal with the path condition, which is the main vehicle for guiding input generation. Our approach *infers an approximate path condition after running the program* while traditional concolic testing collects an exact path condition during program execution. As a simple example, consider the following program: `foo(int x) {if (x > 0) {...}}` Suppose we want to infer a path condition `x>0` without collecting the actual one during the program execution. Suppose two different inputs `x=-1` and `x=1` are generated. After running the program with these inputs, we observe that two executions explore two different paths. Based on this observation, we can infer that `x>0` is a feasible candidate for a path condition. Our technique infers such conditions by using an off-the-shelf inductive program synthesis tool such as Duet [23]. A big advantage of our approach is that it does not require heavyweight instrumentation like traditional symbolic execution for evaluating symbolic expressions of each program variable during the whole execution. For our approach, lightweight instrumentation is sufficient for inferring path conditions because all we need to check is which branches are taken like greybox fuzzing.

Another key idea of our approach is that it *refines* approximate path conditions towards more precise ones in subsequent fuzzing iterations. Unlike concolic testing where a generated input executes exactly the same path for solved constraints, in our approach, a generated input may execute a different path for solved constraints due to potential inaccuracy induced by approximation. Based on observations from the previous and current executions, our technique keeps refining the path conditions and exploring new paths.

We implemented our approach in a tool called PATHFINDER. We conduct extensive experiments on popular real-world DL libraries, PyTorch, and TensorFlow. Our results show that PATHFINDER outperforms existing API-level DL library fuzzers by achieving 67% more branch coverage on average; up to 63% higher than TitanFuzz and 120% higher than FreeFuzz. PATHFINDER detected 61 new bugs of which 59 were confirmed by developers.

In summary, this paper makes the following contributions:

- **Novel Technique.** We propose a novel lightweight concolic testing technique for deep learning (DL) libraries. Our work is the first concolic testing approach for generating DL API inputs by inferring approximate path conditions using program synthesis.
- **Thorough Experiments.** We implement our technique in a tool called PATHFINDER and perform an extensive evaluation with PyTorch and TensorFlow, demonstrating its effectiveness in achieving high code coverage and detecting previously unknown bugs.
- **Replication Package.** The artifact for this work, including the implementation and experimental data, is publicly available on GitHub[1] and Zenodo[2].

II. Background and Motivation

A. Deep Learning Libraries

Figure 1 shows a typical layered structure of a DL library. Its core parts—which are called kernels—are implemented in low-level languages like C++. Most users of DL libraries access these kernels through high-level APIs, which are provided in either Python or C++. Figure 2 shows an example of an API, `affine_grid`, in Python and C++. These APIs perform the same input validation checks as shown in Figure 2, then call the kernel functions when the input is valid. Python APIs and their C++ counterparts, in principle, implement the same functionalities, as illustrated in the example code. In this work, we use C++ APIs for testing to perform a streamlined code instrumentation of both the kernel code and the C++ APIs. It is worth noting that our approach is also applicable to Python APIs through code instrumentation of Python APIs.

B. Concolic Testing

Symbolic execution executes the program with symbolic input values, instead of concrete input values. It maintains a symbolic state, which maps variables having non-concrete values to symbolic expressions, and a path condition PC, which is a first-order formula over symbolic expressions. Both the map and PC are updated during the symbolic execution. At the end of the symbolic execution, solving the PC using a constraint solver generates a test input that executes the same path as the symbolic execution.

[1]https://github.com/starlab-unist/pathfinder-artifact
[2]https://doi.org/10.5281/zenodo.14753432

```
1  void conv2d (Tensor input, Tensor weight,
2               int* padding, int* dilation, int groups) {
3    // TORCH_CHECK(e): Passes if e is true;
4    //                 otherwise, throws an exception.
5    TORCH_CHECK(/* b1 */ input.dim() == 4);
6    TORCH_CHECK(/* b2 */ input.dim() == weight.dim());
7    bool forward_checked = weight.size(0) >= groups &&
8      weight.size(0) % groups == 0 &&
9      input.size(1)==weight.size(1) * groups
10   if (forward_checked) {
11     bool kernel_size_correct =
12       input.size(2) + 2*padding[0]
13         >= dilation[0] * (weight.size(2)-1) + 1 &&
14       input.size(3) + 2*padding[1]
15         >= dilation[1] * (weight.size(3)-1) + 1;
16     if (kernel_size_correct)
17       compute_conv2d(input, weight, padding, dilation);
18  } ... }
```

(a) Simplified code snippet of `conv2d` API

$$\begin{array}{ll}
c_1: & \texttt{input\_rank} = 4 \\
c_2: & \wedge\ \texttt{input\_rank} = \texttt{weight\_rank} \\
c_3: & \wedge\ \texttt{weight\_dim0} \geq \texttt{groups} \\
c_4: & \wedge\ \texttt{weight\_dim0}\ \%\ \texttt{groups} = 0 \\
c_5: & \wedge\ \texttt{input\_dim1} = \texttt{weight\_dim1} \times \texttt{groups} \\
c_6: & \wedge\ \texttt{input\_dim2} + 2 \times \texttt{padding0} \geq \texttt{dilation0} \times (\texttt{weight\_dim2} - 1) + 1 \\
c_7: & \wedge\ \texttt{input\_dim3} + 2 \times \texttt{padding1} \geq \texttt{dilation1} \times (\texttt{weight\_dim3} - 1) + 1
\end{array}$$

(b) Exact PC (path condition) for Line 17

$$\begin{array}{ll}
c_a: & \texttt{input\_rank} = 4 \\
c_b: & \wedge\ \texttt{weight\_rank} = 4 \\
c_c: & \wedge\ \texttt{weight\_dim0} \geq \texttt{groups} \\
c_d: & \wedge\ \texttt{input\_dim0} \geq \texttt{groups} \\
c_e: & \wedge\ 3 - \texttt{input\_dim1} \geq \texttt{groups} + \texttt{weight\_dim1} \\
c_f: & \wedge\ \texttt{TRUE} \\
c_g: & \wedge\ \texttt{input\_dim3} \geq \texttt{weight\_dim3}
\end{array}$$

(c) Approximate PC inferred by our tool for Line 17

Fig. 3: Motivating Example

Concolic testing, also called dynamic symbolic execution, maintains both a concrete state and a symbolic state. It executes a program starting with some random concrete input and collects path conditions PC_1 at conditional statements along the execution path taken by this input. At the end of this concolic execution, by negating an individual branch condition in PC_1, a new path condition PC_2 can be obtained. Then, solving PC_2 using a constraint solver generates a test input that follows a different program path. This process is repeated systematically until all execution paths are explored, or the time budget expires.

C. Motivating Example

This section motivates our technique by illustrating why approximate path conditions inferred from our approach are beneficial for testing DL libraries concolically. Figure 3a presents partial code of a C++ PyTorch API `torch.nn.functional.conv2d`. Suppose we want to test a statement at Line 17. To reach Line 17, we need an input that satisfies complex path conditions presented in Figure 3b.

Existing blackbox DL API fuzzers [8], [9], [10], [11], [13], [14], [24] struggle to generate such inputs because they do not examine the source code during input generation.

Approaches using constraints extracted from documentation [11] or source code [12] are also insufficient as a solution, as they rely on manually defined extraction rules, which often lead to incomplete constraints.

A traditional concolic testing technique would collect the exact path conditions expressed with symbolic input values (e.g., `input_rank`, `weight_dim0`, etc.) shown in Figure 3b. However, even though this exact path condition can be obtained by a symbolic execution engine, concolic testing may struggle to solve non-linear conditions such as c_4, c_5, c_6, and c_7 in Figure 3b; Concolic testing may not be able to generate input that explores this path. Moreover, the full path condition collected beyond Line 17 would be much more complex in real-world DL libraries. For this reason, adopting concolic testing to DL libraries is challenging and has never been addressed in previous research.

Our technique successfully adopts concolic testing to DL libraries by reducing the heavy overhead required to collect exact path conditions. Instead of extracting exact path conditions shown in Figure 3b, our technique infers path conditions based on execution results, as demonstrated in Figure 3c. While some synthesized path conditions, such as c_a to c_c, are inferred precisely, others are often approximate like those from c_d to c_g due to our data-driven path-condition inference. Nevertheless, obtained approximate path conditions can still be effective in guiding path exploration as will be shown in the next section and throughout the paper.

III. Overview

Our approach, PathFinder, guides the exploration of execution paths using path conditions, similar to symbolic execution. However, beyond this similarity, PathFinder differs as described below.

A. Inductively Learning Path Conditions

In symbolic execution, path conditions are *deductively extracted*. For example, when an if-conditional expression, `if (x > y)`, is symbolically executed, the current path condition π is updated into $\pi \wedge (\texttt{x > y})$ or $\pi \wedge \neg(\texttt{x > y})$, depending on which branch is taken. More generally, the rules for updating path conditions are pre-defined for each operation of the programming language, and at runtime, these rules are deductively applied.

While the usefulness of symbolic execution has been shown extensively in the literature, deductively constructing path conditions also imposes several issues such as *incomplete applicability*—e.g., path-condition update rules may not be available for all operations (e.g., system calls)—and *high computational overhead*—updating path conditions typically involves running instrumented code as in CREST [20] or using a dedicated interpreter as in KLEE [19]. In contrast, our approach avoids the aforementioned issues by *inductively learning* path conditions from a set of all inputs executed so far. Below, we describe how we learn path conditions.

Given an execution path π defined as a sequence of branches $b_1 \cdot b_2 \ldots \cdot b_n$ taken during the execution of a program, its path condition π is represented as $\bigwedge_{k=1}^{n} c_k[\![b_k]\!]$ where $c_k[\![b_k]\!]$ denotes the condition for branch b_k learned as described below.

Suppose we have a set of inputs $\mathbb{I}_{pos}$ and $\mathbb{I}_{neg}$ whose execution paths share the same prefix $b_1 \cdot b_2 \ldots \cdot b_{k-1}$ but diverge afterward; that is, every input in $\mathbb{I}_{pos}$ executes $b_1 \cdot b_2 \ldots \cdot b_{k-1} \cdot b_k$, while every input in $\mathbb{I}_{neg}$ executes $b_1 \cdot b_2 \ldots \cdot b_{k-1} \cdot \bar{b}_k$ where $\bar{b}$ denotes the branch that represents the opposite direction of b. Under this setting, PATHFINDER learns $c_k[\![b_k]\!]$ by synthesizing a boolean condition that separates $\mathbb{I}_{pos}$ (i.e., positive inputs) and $\mathbb{I}_{neg}$ (i.e., negative inputs). Thus, $c_k[\![b_k]\!]$ is evaluated to `TRUE` for all inputs in $\mathbb{I}_{pos}$ and `FALSE` for all inputs in $\mathbb{I}_{neg}$.

Due to the inductive nature of our approach, an obtained path condition is likely to be only an *approximation* of the actual path condition. However, as will be explained shortly, these approximate path conditions are still useful in guiding the exploration of execution paths.

B. Path Exploration Guided by Approximate Path Conditions

Consider testing the `conv2d` function shown in Figure 3a. Suppose that for an initial random input I_1, the first parameter, `input`, is of the `Tensor` type with rank 4 (thus, `input.dim()` is 4), and the second parameter, `weight`, is also a `Tensor` but with rank 1 (thus, `weight.dim()` differs from `input.dim()`). Given this input, the execution path takes the if-branch of the first `TORCH_CHECK` statement at Line 5 (i.e., b_1 is true) and the else-branch of the second `TORCH_CHECK` statement at Line 6 (i.e., b_2 is false). That is, I_1 takes the path $b_1 \cdot \bar{b}_2$. At this point, only one input I_1 is available for learning, and as a result, a typical inductive synthesizer such as Duet [23] would synthesize `TRUE` as for $c_1[\![b_1]\!]$ and $c_2[\![\bar{b}_2]\!]$. As a result, we obtain `TRUE` $\wedge$ `TRUE` as the approximate path condition $\hat{\pi}$ for path $b_1 \cdot \bar{b}_2$. We will use notation $\hat{\pi}$ to denote an approximate path condition and π to denote a usual non-approximate path condition.

Unlike concolic execution, we do not negate a conjunct of the path condition to guide the exploration. Instead, we perform a different path exploration strategy as described below. In our approach, an approximate path condition may represent multiple execution paths, as is evident in our running example; currently, we have only one approximate path condition, `TRUE`, capturing all execution paths. To proceed, PATHFINDER chooses an approximate path condition $\hat{\pi}$ from the available ones and generates an input that satisfies $\hat{\pi}$. *This corresponds to exploring the part of the computation tree satisfying* $\hat{\pi}$. From the perspective of fuzzing, it can also be viewed as fuzzing the sub-input space satisfying $\hat{\pi}$.

In our example, suppose that PATHFINDER generates input I_2, where the tensor `input` has a rank of 3, which trivially satisfies the current approximate path condition of `TRUE`. Given this generated input, the execution path takes the else-branch of the first `TORCH_CHECK` statement at Line 5 (i.e., b_1 is false). Based on I_1 and I_2 taking different branches of b_1, the inductive synthesizer refines $c_1[\![b_1]\!]$ from `TRUE` to a stronger condition, say, `input_rank > 3`. We now have two approximate path conditions, $c_1[\![b_1]\!] \wedge c_2[\![\bar{b}_2]\!]$ and $\neg c_1[\![b_1]\!]$, which are simplified in our example to `input_rank > 3` and `input_rank ≤ 3`, respectively. We then generate the next input that satisfies either of the two approximate path conditions and repeat the process.

As the process continues, PATHFINDER refines the approximate path conditions with increasing accuracy for the following two reasons. First, both positive and negative inputs will be available for more branches, which will have the synthesizer produce a non-`TRUE` condition for those branches. Second, as more examples will be available for each branch, the synthesizer will be able to produce more precise conditions.

C. Comparison with Other Path Exploration Techniques

We here summarize our approach by comparing it with other path exploration techniques such as symbolic execution and fuzzing. In symbolic execution, each feasible path condition identifies a distinct execution path. Thus, every generated input is guaranteed to take a new execution path. However, this guarantee comes at the cost of high computational overhead. To extract a path condition, symbolic execution typically requires heavyweight code instrumentation or a custom interpreter. In comparison, PATHFINDER requires only lightweight instrumentation to monitor branch directions during execution. In exchange for faster execution, PATHFINDER's inductive learning may not produce precise path conditions. In other words, PATHFINDER *trades off the precision of path conditions for the efficiency of extracting them.*

Our efficiency-over-precision approach is akin to that of fuzzing. However, unlike typical fuzzing, PATHFINDER *uses learned path conditions to guide the exploration of execution paths.* It can be viewed that we divide the input space into subspaces based on the learned path conditions; each subspace corresponds to a distinct approximate path condition.

Overall, PATHFINDER sits between symbolic execution and fuzzing. Compared to symbolic execution, PATHFINDER generates inputs at a faster pace. While a fuzzer can generate inputs more quickly than PATHFINDER since PATHFINDER invokes an SMT solver to generate new inputs, PATHFINDER can explore execution paths more effectively by using learned path conditions. This balance between efficiency and effectiveness is the key strength of PATHFINDER. At the early phase of exploration, PATHFINDER can quickly generate inputs covering new paths without the need for precise path conditions. At the later phase, PATHFINDER can generate inputs that cover new paths effectively by using learned path conditions whose precision has been improved as more inputs are collected.

IV. METHODOLOGY

In this section, we describe the algorithm of PATHFINDER (§ IV-A) and its optimizations (§ IV-B).

A. Algorithm

Algorithm 1 shows the algorithm of PATHFINDER. It takes as input a target function to test and returns a bug revealing input found in the given time budget. Given the target function f, PATHFINDER automatically generates the precondition φ of f based on the type information of each parameter. For example, if a parameter has a tensor type, precondition φ

Algorithm 1 PATHFINDER Algorithm

```
Input: a target function $f$
Output: a bug revealing input $i$; NONE if not found
 1: // Step 1: Initialization
 2: $\varphi \leftarrow$ PRECONDGEN($f$) // $f$'s precondition is automatically generated based on the type of each parameter
 3: $\mathbb{I} \leftarrow \varnothing$ // A set of executed inputs
 4: $\hat{\Pi} \leftarrow \{$TRUE$\}$ // A set of approximate path conditions
 5:
 6: while $time\_elapsed < time\_out$ do
 7:     // Step 2: Input Generation
 8:     $\hat{\pi} \leftarrow$ CHOOSE($\hat{\Pi}$)
 9:     $i \leftarrow$ GENINPUT($\varphi \wedge \hat{\pi}$) // $i \models \varphi \wedge \hat{\pi}$
10:
11:     // Step 3: Running $f$ with input $i$
12:     // $\mathcal{B}$: a sequence of executed branches
13:     $(\mathcal{B}, crashed) \leftarrow$ RUN$[\![f]\!](i)$
14:     if $crashed$ = TRUE then
15:         return $i$
16:     end if
17:
18:     // Step 4: Refining $\hat{\Pi}$
19:     $\hat{\Pi} \leftarrow$ REFINE($\mathcal{B}, \mathbb{I} \cup \{i\}, \hat{\Pi}$)
20:     $\mathbb{I} \leftarrow \mathbb{I} \cup \{i\}$
21: end while
22: return NONE
```

includes a constraint that all tensor dimensions should be positive. More details are provided in Section V-A.

While running, PATHFINDER maintains a set of approximate path conditions defined as follows:

Definition 1 (Approximate Path Condition): Consider an execution path following a sequence of branches $\mathcal{B}$. Then, the approximate path condition of this execution path is:

$$\bigwedge_{k=1}^{n} c_k[\![b_k]\!]$$

where $c_k[\![b_k]\!]$ represents the inductively inferred branch condition of the k-th branch $b_k \in \mathcal{B}$, and n is the length of $\mathcal{B}$. $c_k[\![b_k]\!]$ should satisfy the following properties:

1) $c_k[\![b_k]\!]$ should be expressed as a boolean formula over formal parameters of the target function.

2) Consider a set of all inputs $\mathbb{I}$ executed so far and its two subsets $\mathbb{I}_{pos}$ and $\mathbb{I}_{neg}$ satisfying the following. $\mathbb{I}_{pos}$ contains all inputs in $\mathbb{I}$ that executes a sequence of branches, $b_1 \cdot b_2 \cdot \ldots \cdot b_{k-1} \cdot b_k$. Meanwhile, $\mathbb{I}_{neg}$ contains all inputs in $\mathbb{I}$ that executes a sequence of branches, $b_1 \cdot b_2 \cdot \ldots \cdot b_{k-1} \cdot \bar{b}_k$, where $\bar{b}$ denotes the branch that represents the opposite direction of b. Then, for all inputs i_p in $\mathbb{I}_{pos}$, applying $c_k[\![b_k]\!]$ to i_p should return TRUE, and for all inputs i_n in $\mathbb{I}_{neg}$, applying $c_k[\![b_k]\!]$ to i_n should return FALSE. Note that there can exist only $\mathbb{I}_{pos}$ without $\mathbb{I}_{neg}$. In this case, $c_k[\![b_k]\!] \equiv$ TRUE.

In the beginning, PATHFINDER initializes the set of approximate path conditions as $\hat{\Pi} = \{$TRUE$\}$ (Line 4). Once PATHFINDER enters the main loop (Lines 6-21), PATHFINDER repeats the following three steps: (1) input generation, (2) running the target function with the generated input, and (3) refining a set of approximate path condition, $\hat{\Pi}$, based on the execution result. We below describe each step.

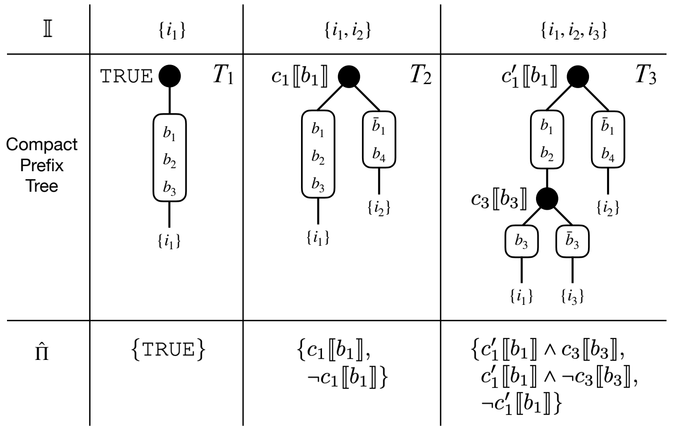


Fig. 4: Efficiently refining approximate path conditions using compact prefix tree. $\mathbb{I}$ and $\hat{\Pi}$ represent the set of executed inputs and the set of approximate path conditions, respectively.

1) Input Generation: To generate the next input to execute, we choose an approximate path condition $\hat{\pi}$ from the current $\hat{\Pi}$ (Line 8). In the current implementation, we randomly select an approximate path condition while using other selection strategies should be possible. We then use an SMT solver to generate an input i that satisfies $\hat{\pi}$ and the precondition φ of the target function (Line 9).

2) Running the Target Function: In the next step, we run the target function f with the generated input i (Line 13). While executing f, we record the sequence of executed branches $\mathcal{B}$. If f crashes, we return the crashing input i.

3) Refining the Approximate Path Conditions: Given the sequence of branches $\mathcal{B} \equiv b_1 \cdot b_2 \cdot \ldots \cdot b_n$ executed with the input i, we refine the set of approximate path conditions $\hat{\Pi}$ (Line 19). We iterate over each branch b_k in $\mathcal{B}$ and construct $\mathbb{I}_{pos}$ and $\mathbb{I}_{neg}$ as described in Definition 1. We then synthesize a boolean formula $c_k[\![b_k]\!]$ that separates $\mathbb{I}_{pos}$ from $\mathbb{I}_{neg}$. This way, we obtain an approximate path condition for $\mathcal{B}$ and add it to $\hat{\Pi}$.

We also refine the existing approximate path conditions in $\hat{\Pi}$. Consider an approximate path condition $\hat{\pi} \in \hat{\Pi}$ defined as $\bigwedge_{k=1}^{m} c_k[\![b'_k]\!]$. We refine $c_k[\![b'_k]\!]$ if for all j such that $1 \leq j \leq k-1$, b'_j is identical to $b_j \in \mathcal{B}$. That is, the approximate path condition $\hat{\pi}$ and the new input i follow the same branches up to the $(k-1)$-th branch. In this case, we refine $c_k[\![b'_k]\!]$ to $c_k[\![b_k]\!]$, a new branch condition inferred from $\mathbb{I}_{pos}$ and $\mathbb{I}_{neg}$. We efficiently perform the aforementioned task as below.

Efficient Data Structure. To efficiently refine approximate path conditions, we represent the set of approximate path conditions $\hat{\Pi}$ using a compact prefix tree, also known as a radix tree. Consider an example from Figure 4. It presents a scenario where PATHFINDER generates three inputs, i_1, i_2, and i_3 in sequence; see the first row. The second row illustrates, for each input, how we refine the compact prefix tree (i.e., $T_1 \rightarrow T_2 \rightarrow T_3$). The last row presents a set of approximate path conditions represented by the compact prefix tree.

Suppose when the target function is run with input i_1, it follows the execution path, $b_1 \cdot b_2 \cdot b_3$. Since we have only one execution path, the compact prefix tree T_1 has only one root node (●) and a leaf node (i.e., $\{i_1\}$). These two nodes are connected by an edge labeled with the sequence of branches, $b_1 \cdot b_2 \cdot b_3$. Suppose the next input, i_2, deviates from b_1 and takes an opposite direction, $\bar{b}_1$, followed by b_4. We add a new edge labeled with $\bar{b}_1 \cdot b_4$. Since we now have both positive and negative examples for branch b_1, the synthesizer can infer a non-TRUE condition, $c_1[\![b_1]\!]$. In the running example, the updated compact prefix tree T_2 has now two paths whose approximate path conditions are $c_1[\![b_1]\!]$ and $\neg c_1[\![b_1]\!]$, respectively. Now suppose that we generate the next input, i_3, satisfying $\neg c_1[\![b_1]\!]$, expecting to execute the path $\bar{b}_1 \cdot b_4$. However, due to the imprecision of the approximate path condition, the use of i_3 may lead to a different execution path, $b_1 \cdot b_2 \cdot \bar{b}_3$. Accounting for this counter-example, PATHFINDER refines $c_1[\![b_1]\!]$ to $c_1'[\![b_1]\!]$ which separates the positive examples, i_1 and i_3, from the negative example, i_2. Also, a new branch condition $c_3[\![b_3]\!]$ is inferred based on i_1 and i_3.

B. Other Optimizations

We here describe the other optimization strategies used in PATHFINDER to improve the efficiency of the testing process.

1) Efficient Use of the Synthesizer: Many modern inductive program synthesizers, including Duet [23], are syntax-guided—i.e., they synthesize programs that follow a given grammar—and examples-based—i.e., they synthesize programs that satisfy a set of input-output examples; in our setting, the output is a boolean value indicating the direction of the branch. For efficient use of the synthesizer, we control the grammar and the number of examples used for synthesis, as described below.

Staged Synthesis. Duet is a syntax-guided synthesizer; all synthesized conditions should follow the given grammar. In general, the more complex the grammar, the more time the synthesizer takes to find a solution. Using simple grammar can also be beneficial for input generation, as finding a solution for a simple path condition is easier. Considering these, we initially use a simple grammar shown in Figure 5a. Only if synthesis fails with the simple grammar, we use the more complex grammar shown in Figure 5b.

Tolerant Branch Condition Refinement. Our main goal is to efficiently explore execution paths to find bugs, rather than to synthesize precise branch conditions. As will be shown in our experimental results, using approximate branch conditions is often sufficient for exploring new paths and finding bugs. To avoid spending too much time on synthesizing precise conditions, we refrain from synthesizing conditions when the current approximate path condition is already accurate enough. Consider a branch condition $c_k[\![b_k]\!]$ synthesized from positive inputs $\mathbb{I}_{pos}$ and negative inputs $\mathbb{I}_{neg}$. Suppose $c_k[\![b_k]\!]$ is not precise and applying it to a new input i returns TRUE while the actual execution takes the opposite direction. We measure

$$
\begin{array}{rcl@{\qquad}rcl}
\mathit{Cond}_0 & \to & \mathit{Cond}_1 & \mathit{Const} & \to & n \in \{0,1,2,\cdots\} \\
 & \mid & \neg\mathit{Cond}_1 & \mathit{Var} & \to & \mathtt{x} \mid \mathtt{y} \mid \cdots \\
\mathit{Cond}_1 & \to & \mathit{Var} = \mathit{Const} & & & \\
 & \mid & \mathit{Var} = \mathit{Var} & & &
\end{array}
$$

(a) Level-1 Grammar

$$
\begin{array}{rcl@{\qquad}rcl}
\mathit{Cond}_0 & \to & \mathit{Cond}_1 & \mathit{Expr}_1 & \to & \mathit{Const} \\
 & \mid & \mathit{Cond}_1 \wedge \mathit{Cond}_1 & & \mid & \mathit{Var} \\
 & \mid & \mathit{Cond}_1 \vee \mathit{Cond}_1 & & \mid & \mathit{Const} \times \mathit{Var} \\
 & \mid & \neg\mathit{Cond}_1 & & \mid & \mathit{Var} \div \mathit{Const} \\
\mathit{Cond}_1 & \to & \mathit{Expr}_0 = \mathit{Expr}_0 & & \mid & \mathit{Var}\ \%\ \mathit{Const} \\
 & \mid & \mathit{Expr}_0 < \mathit{Expr}_0 & \mathit{Const} & \to & n \in \{0,1,2,\cdots\} \\
\mathit{Expr}_0 & \to & \mathit{Expr}_1 & \mathit{Var} & \to & \mathtt{x} \mid \mathtt{y} \mid \cdots \\
 & \mid & \mathit{Expr}_0 + \mathit{Expr}_0 & & & \\
 & \mid & \mathit{Expr}_0 - \mathit{Expr}_0 & & &
\end{array}
$$

(b) Level-2 Grammar

Fig. 5: Grammar for branch conditions

the accuracy of $c_k[\![b_k]\!]$ using $\mathbb{I}_{pos}$ and $\mathbb{I}_{neg} \cup \{i\}$, using MCC (Matthews correlation coefficient). Only if the accuracy is below a threshold (we empirically use 0.6), we refine $c_k[\![b_k]\!]$.

Sampling Examples. Another factor that affects the synthesis time is the number of examples used for synthesis. To reduce the synthesis time, we set a limit N on the number of examples used for synthesis. In our experiments, we set N to 30.

2) Nondeterministic Branch Pruning: When exploring the execution paths based on path conditions, an underlying assumption is that the executed branches are deterministic. However, this is not always the case. For example, consider `rand()>0`. Including such nondeterministic branches in the path condition can deteriorate the performance of the synthesizer, as demonstrated in § VI-B.

We remove nondeterministic branches from the compact prefix tree. For example, in Figure 4, if b_2 is nondeterministic, we remove it from the compact prefix tree. To avoid incurring an extra cost of checking nondeterminism, we identify nondeterministic branches only opportunistically when the same input is generated again during the testing process; if different subsequences of branches are observed between two executions, we identify nondeterministic branches using a variant of Myers' algorithm [25] and prune them from the approximate path condition.

3) Diverse Input Generation: Generating diverse input is crucial when performing testing. PATHFINDER generates an input satisfying the inferred approximate path condition $\hat{\pi}$. While we need to generate diverse input satisfying $\hat{\pi}$, an SMT solver such as Z3 we use often generates the same input when the same $\hat{\pi}$ is given. To obtain more diverse inputs, we strengthen $\hat{\pi}$ into $\hat{\pi} \wedge \psi$ where ψ is a random constraint. We construct ψ with a template $x \oplus y$, where x and y are randomly chosen from input parameters, and $\oplus$ is randomly selected from $\{=, \neq, <, \leq\}$.

V. EXPERIMENTAL SETUP

To evaluate our technique PATHFINDER, we investigate the following three research questions:

TABLE I: Type-based precondition generation rules

Param	***Internal Param***	***Precondition***
`int n`	`n`	`INT_VAL_MIN` ≤ `n` ≤ `INT_VAL_MAX`
`float r`	`r`	`r` ∈ {0, 1, ..., `FLOAT_VAL_MAX`}
`Tensor t`	`t_dtype`	`DT_MIN` ≤ `t_dtype` ≤ `DT_MAX`
	`t_rank`	0 ≤ `t_rank` ≤ 5
	`t_dim0`	1 ≤ `t_dim0` ≤ `INT_VAL_MAX`
	...	...
	`t_dim4`	1 ≤ `t_dim4` ≤ `INT_VAL_MAX`

• RQ1: How effective is PATHFINDER in achieving code coverage?
• RQ2: How do various optimization strategies of PATHFINDER affect its performance?
• RQ3: How effective is PATHFINDER in detecting bugs?

A. Implementation

We implemented PATHFINDER in C++. Our implementation includes an automatic test driver generator; given a target API f of PyTorch or TensorFlow, our tool generates a test driver that invokes f with the inputs generated by PATHFINDER. While automatically generating a test driver for an arbitrary function is challenging, we found that supporting PyTorch/TensorFlow APIs is feasible. To synthesize branch conditions, we use an inductive program synthesizer, Duet [23]. For input generation, we use Z3 SMT solver [26].

Precondition Generation. When testing a target API f, the inputs generated by PATHFINDER should satisfy the preconditions of f, as we described in § IV-A. Depending on the type information of f's parameters, we enforce different constraints on the inputs, as shown in Table I. For example, given an integer type parameter `n`, we enforce that `n`'s value should be within the range of `INT_VAL_MIN` and `INT_VAL_MAX`. For a floating-point type parameter `r`, we restrict its value to be one of the predefined floating-point constants. This is to expedite input generation using an SMT solver. For the `Tensor` type, we enforce constraints on the tensor's data type, rank, and dimensions. Once values satisfying these constraints are generated, we put them into a `Tensor` object and pass it to the target API.

B. Baseline Tools

We compare PATHFINDER with five existing techniques: FreeFuzz [8], DeepREL [13], TitanFuzz [9], ACETest [12], and IvySyn [24]. We selected FreeFuzz, DeepREL, and TitanFuzz because they are state-of-the-art API-level fuzzing tools for DL libraries like PATHFINDER. Although a recent work FuzzGPT [10] is also an API-level fuzzer, we could not include it in our evaluation because the tool is not publicly available. We also included ACETest and IvySyn, which target kernel functions of DL libraries. These tools require test drivers for kernel functions and we use PATHFINDER's test driver generator to prepare them.

C. DL Libraries and APIs

We consider both PyTorch [1] (v2.2) and TensorFlow [2] (v2.16) since they are the two most popular DL libraries and are widely studied in the literature. For comparison with IvySyn, we use PyTorch v1.11, which is supported by IvySyn's replication package because it does not support PyTorch v2.

TABLE II: Number of target APIs for PATHFINDER and the baselines. PATHFINDER* denotes target APIs selected for comparison with the baselines.

	PATHFINDER	PATHFINDER*	FreeFuzz	DeepREL	TitanFuzz	ACETest	IvySyn
PyTorch	946	661	585	585	649	379†	340†
TensorFlow	517	517	135	137	273	486	-

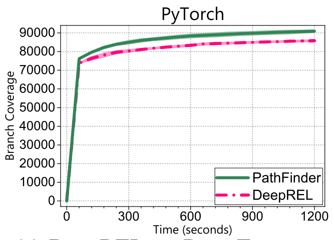
Target APIs. Table II lists the number of target APIs used in our experiments. Our test driver generators successfully create test drivers for 946 C++ APIs in PyTorch and 517 C++ APIs in TensorFlow, respectively. For the comparison evaluation with the baselines, we selected 661 PyTorch and 517 TensorFlow APIs (denoted as "PATHFINDER*" in Table II) having exact Python counterpart APIs (e.g., `torch::nn::functional::max_pool1d` in C++ and `torch.nn.functional.max_pool1d` in Python). We obtained each baseline's target APIs from its replication package. For a fair comparison, we extracted those *common* APIs shared by both "PATHFINDER*" and the baseline. For example, as shown in Table II, we collected 585 common PyTorch APIs shared between PATHFINDER*'s and DeepREL's target APIs. For FreeFuzz-PyTorch, we used the same set of target APIs from DeepREL because it shares the same fuzzing engine as FreeFuzz and supports a larger set of APIs.

For ACETest and IvySyn, their PyTorch target functions indicated by † in Table II are internal kernel functions rather than high-level APIs. We collected the kernel functions used in the replication packages of ACETest and IvySyn and considered those functions for which our driver generator could successfully produce drivers. In RQ1, out of IvySys's 340 targets, we used 241 functions that terminate without a crash because IvySyn cannot measure coverage when a crash occurs. Finally, we excluded TensorFlow for IvySyn because its target internal functions lack the type information required by our driver generator.

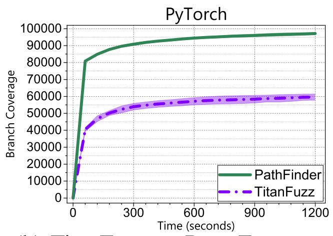
D. Experimental Environment

Environment. We conducted our experiments on two machines. For the PyTorch experiment, we used a machine equipped with Intel Xeon Platinum 8468 CPU and 4 NVIDIA RTX A6000 GPUs. For the TensorFlow experiment, we used a machine with AMD EPYC 7763 CPU and 8 NVIDIA RTX A6000 GPUs. We also ensured each fuzzing task was allocated a single CPU core during the experiments.

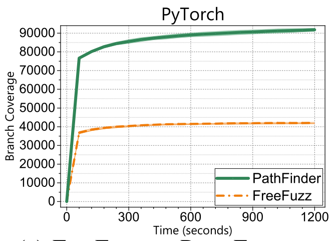
Fuzzing budget. By default, we use a 20-minute fuzzing budget per target API. Unlike PATHFINDER that does not require any preparation process before fuzzing, all baseline techniques except IvySyn require additional preparation time. This includes gathering seed inputs (for FreeFuzz, DeepREL, and TitanFuzz) or extracting input constraints (for ACETest). In this experiment, we exclude such preparation time for all baselines from the time budget. Meanwhile, we conduct the coverage analysis in RQ1 and the ablation study in RQ2 five times and report the average number.



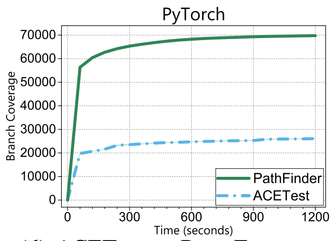
(a) DeepREL vs PATHFINDER



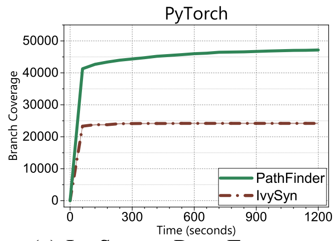
(b) TitanFuzz vs PATHFINDER



(c) FreeFuzz vs PATHFINDER



(d) ACETest vs PATHFINDER



(e) IvySyn vs PATHFINDER

Fig. 6: Results of RQ1 for PyTorch

Bug Detection. We manually analyzed each detected bug and identified the unique ones. In particular, we focus on finding crash bugs in our experiments.

VI. EXPERIMENTAL RESULTS

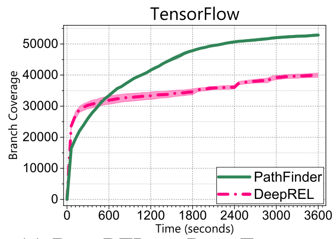
A. RQ1: Branch Coverage Analysis

To evaluate the effectiveness of PATHFINDER in exploring diverse program paths, we measure branch coverage, a common metric for evaluating the effectiveness of test generation tools. To ensure a fair comparison without bias towards any specific API language (i.e., C++ or Python), we measure coverage of only C++ kernel codes.

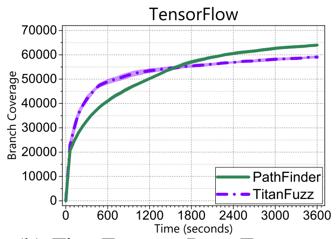
The results are shown in Figures 6 and 7 where we compare PATHFINDER with the five baselines mentioned in §V-B: FreeFuzz, DeepREL, TitanFuzz, ACETest, and IvySyn. The X-axis shows the elapsed time in seconds, and the Y-axis shows the number of covered branches across all target APIs. A coordinate (x, y) indicates that at time x, y branches are covered across all target APIs. We repeat the experiment five times for each target API and present the mean results in the figure, along with 95% confidence intervals illustrated as shades around the mean line.

As for the time budget, we set it to 20 minutes for each pair of a tool and an API of PyTorch. For TensorFlow, we could not identify a clear winner in the initial 20 minutes, so we extended the time budget to 60 minutes.

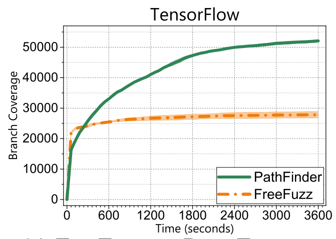
Our results show that PATHFINDER substantially outperforms all existing state-of-the-art tools we compare with. We below discuss notable observations from the results.



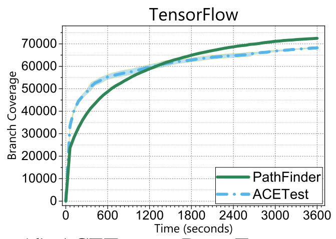
(a) DeepREL vs PATHFINDER



(b) TitanFuzz vs PATHFINDER



(c) FreeFuzz vs PATHFINDER



(d) ACETest vs PATHFINDER

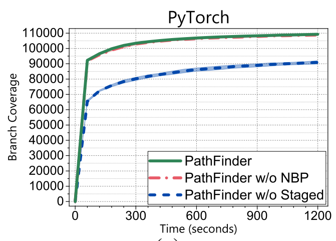
Fig. 7: Results of RQ1 for TensorFlow. IvySyn is excluded for TensorFlow as discussed in § V-C.

PyTorch vs. TensorFlow. While PATHFINDER outperforms all baselines, its early-stage performance for TensorFlow is not as good as for PyTorch. This seems related to the fact that TensorFlow uses nondeterministic branches more frequently than PyTorch. The average number of nondeterministic branches found in PyTorch APIs is 103.4, while in TensorFlow APIs, it is 2571.3, which is about 25 times larger. Despite the slow start, PATHFINDER eventually covers more branches than the baselines in TensorFlow as well, and the gap between PATHFINDER and the baselines widens over time.

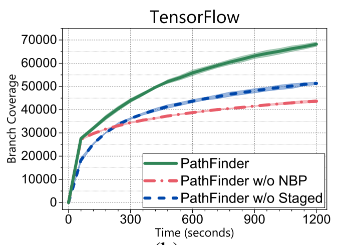
Unit Testing vs. Integration Testing. Unlike PATHFINDER that tests an individual target API, TitanFuzz, using an LLM (Large Language Model), constructs a sequence of API calls that includes the target API. This can be viewed as a form of integration testing. In our experimental results, we account for all executed branches, whether or not they belong to the target API. Nevertheless, PATHFINDER still outperforms TitanFuzz, demonstrating the effectiveness of our technique in exploring diverse execution paths.

Similar to TitanFuzz, DeepREL tests not only the target API (e.g., `AdaptiveAvgPool3d`) but also other APIs (e.g., `AdaptiveMaxPool3d`) that are considered functionally similar to the target API. DeepREL effectively reuses the same input to test multiple APIs. PATHFINDER achieves higher branch coverage than DeepREL, without exploiting such additional information on API similarity. Note that exploiting API similarity and our efficient path exploration algorithm are orthogonal, and combining them could further improve the performance of PATHFINDER.

Fully automatic vs. Semi-automatic. ACETest, similar to PATHFINDER, leverages path constraints for input generation. However, unlike PATHFINDER that infers path condition fully automatically, ACETest relies on manually defined rules for extracting path constraints. Despite these manual efforts required by ACETest, PATHFINDER outperforms for both Py-



(a)



(b)

Fig. 8: Results of RQ2 on our two optimization strategies, nondeterministic branch pruning (NBP) and staged synthesis

Torch and TensorFlow with the performance gap being particularly significant for PyTorch. This gap is mainly due to the limitation of ACETest's incomplete manual rules. For example, its rules cannot handle PyTorch's certain code components (e.g., data type check of a tensor like `tensor.is_complex()`). This leaves many branches uncovered by ACETest. Although adding new rules could potentially increase coverage, doing so would require significant additional manual efforts. In contrast, PathFinder can achieve high coverage without manual efforts.

RQ1: PathFinder achieves higher branch coverage than state-of-the-art DL testing tools, demonstrating the effectiveness of our technique in exploring diverse execution paths.

B. RQ2: Ablation Study

To evaluate the effectiveness of PathFinder's optimization strategies, we conduct an ablation study by comparing PathFinder with two variants: PathFinder without nondeterministic branch pruning (NBP) and PathFinder without staged synthesis. For this ablation study, we use all available target APIs for PyTorch and TensorFlow, i.e., 946 PyTorch APIs and 517 TensorFlow APIs (see Table II) with the 20-minute time budget for each target API.

Note that if we keep nondeterministic branches without pruning, Duet, the synthesizer we use, always fails because the same input appears both in the positive and negative examples. Rather than considering such a clearly inferior variant, we here compare PathFinder with an alternative way to handle nondeterministic branches; instead of pruning a nondeterministic branch, we set `TRUE` to its condition. For example, in Figure 4, if b_1 is nondeterministic, we set $c_1[\![b_1]\!]$ to `TRUE` while keeping its both branches. This is different from the original PathFinder, which removes b_1 from the compact prefix tree.

As for disabling staged synthesis, we experiment with a variant of PathFinder that directly employs Level-2 grammar without attempting Level-1 grammar (refer to Figure 5 for these grammars). Recall that Level-1 grammar is simpler than Level-2 grammar, and it is used in the original PathFinder to quickly find a solution when possible.

Figure 8 shows the results. More branches are covered when PathFinder employs both NBP and staged synthesis than when it does not. As mentioned, TensorFlow APIs have more nondeterministic branches than PyTorch APIs, which explains why NBP is more effective in TensorFlow than in PyTorch.

RQ2: PathFinder performs better when both optimization strategies, nondeterministic branch pruning and staged synthesis, are employed than when either is omitted. This demonstrates the effectiveness of our optimization strategies.

C. RQ3: Bug Detection Analysis

Table III summarizes the statistics of bugs detected by PathFinder. For this experiment, we use all available target APIs for PyTorch and TensorFlow in our benchmark. PathFinder detected 61 bugs, with 59 of them confirmed as previously unknown by developers. Among these confirmed bugs, 32 have been fixed. Two bug reports were rejected as false alarms. Notably, for 21 of the 32 fixed bugs, patches were applied in the kernel functions, which demonstrates that PathFinder effectively explores deep and diverse execution paths.

Besides, Table IV presents the distribution of different types of bugs for all confirmed bugs. Segfault and FPE refer to Segmentation Fault and Floating Point Exception, respectively. The results show that PathFinder detects various types of severe bugs, such as heap buffer overflow and stack overflow, which might also lead to security vulnerabilities.

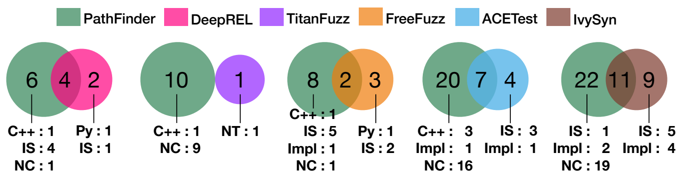
Figure 9 presents the comparison results with the baselines. For the comparison, we consider the set of common target

TABLE III: Summary of bugs detected by PathFinder

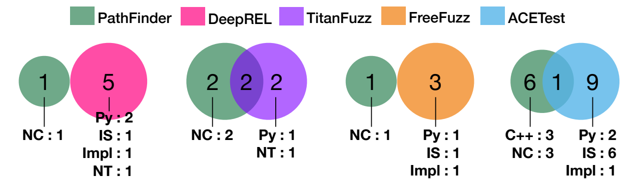
DL Library	Total	Confirmed	Fixed	Rejected
PyTorch	43	41	23	2
TensorFlow	18	18	9	0
Total	61	59	32	2

TABLE IV: Type of bugs detected by PathFinder

DL Library	Bug Type					Total
	Heap Buffer Overflow	Stack Overflow	Segfault	FPE	Internal Assertion	
Pytorch	7	2	9	4	19	41
Tensorflow	1	0	1	0	16	18
Total	8	2	10	4	35	59



(a) PyTorch



(b) TensorFlow

Fig. 9: Bugs detected by PathFinder and baselines

TABLE V: Total number of inputs generated by each tool and their valid input rates.

DL Lib	PyTorch										TensorFlow							
Tool	FreeFuzz	Ours	ACETest	Ours	DeepREL	Ours	TitanFuzz	Ours	IvySyn	Ours	FreeFuzz	Ours	ACETest	Ours	DeepREL	Ours	TitanFuzz	Ours
Total	163,679 K	27,569 K	164,807 K	39,761 K	110,903 K	26,082 K	1,611 K	30,868 K	200,715 K	21,464 K	42,667 K	5,191 K	139,548 K	17,545 K	25,579 K	5,434 K	1,419 K	15,060 K
Valid	24,955 K	15,780 K	98,762 K	25,107 K	98,858 K	15,248 K	1,123 K	17,911 K	194,746 K	10,602 K	36,349 K	1,984 K	49,130 K	3,237 K	20,239 K	2,062 K	886 K	2,899 K
Ratio (%)	15.24	57.24	59.93	63.14	89.14	58.46	69.68	58.02	97.03	49.40	85.19	38.22	35.21	18.36	79.13	37.95	62.46	19.25
Branch	41,850	92,208	26,047	69,772	85,859	91,314	59,917	97,521	24,322	47,114	27,839	51,237	68,200	72,462	40,056	52,181	59,063	63,953

APIs and kernel functions listed in Table II for each baseline. For fair analysis, we use the top frame information of a crash's stack trace. We consider a crash as the same bug if the top frame of that crash is identical to that of the compared tool.

The results of Figure 9 show that PATHFINDER detects significantly more unique bugs than the compared tool—PATHFINDER uniquely detects 76 bugs (66 for PyTorch and 10 for TensorFlow) while it misses only 38 bugs (19 for PyTorch and 19 for TensorFlow).

We further analyze why those unique bugs have been missed by the compared tool (i.e., other tools for PATHFINDER and PATHFINDER for other tools). We classified them into five categories, **NC, C++/Py, IS, Impl, and NT**, and tagged in Figure 9. **NC** happens when the crash point is not covered by the compared tool due to its limited code coverage. This is the most common category for the unique bugs detected by PATHFINDER, which demonstrates PATHFINDER's high path exploration capability. **C++/Py** includes language-specific bugs that are manifested exclusively in either the C++ or Python API. These bugs occur due to the inconsistent implementation between the C++ and Python APIs even though both are designed to function identically in principle, as shown in Figure 2. **IS** refers to those bugs missed by the compared tool due to its limited input space. For instance, despite PATHFINDER's high path exploration capability, it misses bugs arising from edge cases (e.g., extreme values) existing the outside of the input space defined in Table I. Similarly, DeepREL, FreeFuzz, and IvySyn did not generate rarely used inputs in practice (e.g., quantized integers). **Impl** includes those bugs missed by the compared tool due to its limitation of implementation design. For example, PATHFINDER, DeepREL, FreeFuzz, and IvySyn terminate their fuzzing campaign upon encountering a crash, preventing further discovery of additional bugs within the same API. Finally, **NT** refers to bugs located in non-target APIs. As discussed in § VI-A, DeepREL and TitanFuzz construct a sequence of APIs, so PATHFINDER cannot detect bugs that are executed exclusively by the compared tool.

RQ3: PATHFINDER is highly effective in detecting severe bugs that require deep program exploration, particularly within kernel functions, and detects significantly more unique bugs that are not detected by the baselines.

VII. Discussion and Threats to Validity

A. Comparison with Greybox Fuzzing and Concolic Testing

We have shown PATHFINDER's effectiveness for testing DL libraries; it outperforms existing state-of-the-art tools for DL library testing. In this section, we discuss how PATHFINDER compares with greybox fuzzing and concolic testing, two representative non-blackbox testing techniques. For greybox fuzzing, we used libfuzzer [16], a well-known greybox API fuzzer. For concolic testing, we used Eclipser [27], the only concolic-execution-based tool that we could successfully run on PyTorch. Eclipser is a state-of-the-art binary-based fuzzing tool that iterates between concolic testing and greybox fuzzing, outperforming KLEE [19] in terms of code coverage [27]. We ran PATHFINDER, libfuzzer, and Eclipser on 661 C++ APIs of PyTorch (denoted as "PATHFINDER*" in Table II) with a 20-minute time budget.

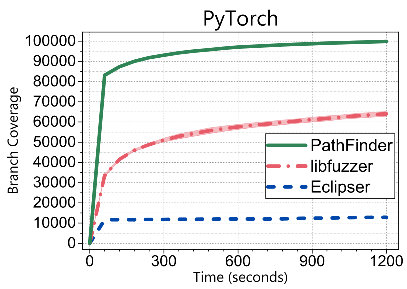


Fig. 10: Branch coverage of PATHFINDER compared with greybox fuzzing (libfuzzer) and concolic testing (Eclipser)

Figure 10 shows the results. PATHFINDER substantially outperforms both Eclipser and libfuzzer. The poor performance of Eclipser seems to be due to the slow runtime, a typical drawback of concolic testing. While libfuzzer performs better than Eclipser, it still lags behind PATHFINDER. This result suggests that PATHFINDER's path exploration guided by path conditions is more effective than the conventional coverage-guided approach of libfuzzer. Despite being guided by coverage feedback, greybox fuzzing struggles to exercise new branches when code coverage does not change over input mutations. This means that coverage feedback is only useful when the mutated input explores a new execution path. In contrast, PATHFINDER generates inputs capable of penetrating an approximate branch condition and exploring new paths.

B. Input Validity

We have shown throughout the paper that PATHFINDER achieves substantially higher branch coverage than state-of-the-art tools. PATHFINDER achieves such outstanding performance despite the fact that our technique is agnostic to input validity; our path-condition synthesis technique does not aim to rule out invalid inputs, but aims to explore diverse paths.

Table V shows (1) how many inputs each tool generated ("Total"), (2) the percentage of valid inputs ("Valid" and "Ratio (%)"), and (3) the number of covered branches ("Branch"). We observe PATHFINDER covers more branches than the other tools even when its valid-input ratio is lower. This result suggests the effectiveness of fuzzing tools is not solely determined by the ratio of valid inputs. PATHFINDER's ability to generate inputs that explore diverse execution paths appears to be a more critical factor in achieving high branch coverage.

C. Threats to Validity

Generality. There is a potential threat to the generality of our approach. To address this, we evaluated PATHFINDER on the two most representative DL libraries: PyTorch and TensorFlow, which are widely used in literature.

Bug detection. When counting new bugs detected by PATHFINDER and the baselines, we ensured all detected bugs were reproducible. Additionally, in our experiments, we only consider crashes as bugs and do not consider non-crash bugs such as computation bugs. These bugs may be detected with differential testing used in prior work [8], [28].

VIII. Related Work

A. Deep Learning Library Fuzzing

API-level fuzzing. FreeFuzz [8] mines usage examples of DL APIs from open source and mutates inputs based on the observed values in the wild. DeepREL [13] and TENSORSCOPE [14] extract related APIs within the same library [13] and counterpart APIs across different libraries [14], respectively, and borrow test inputs from other APIs to generate inputs for target APIs. TitanFuzz [9] and FuzzGPT [10] leverage large language models to generate a valid API sequence involving a target API's invocation. IvySyn [24] performs type-aware mutation-based fuzzing on kernel code.

DocTer [11] and ACETest [12] extract input constraints for each API parameter to generate inputs that pass the input validity checks. They focus on extracting input constraints before the fuzzing phase and perform fuzzing by generating random inputs satisfying those constratins.

Although all the aforementioned techniques have shown effectiveness in testing DL library APIs, they either perform blackbox fuzzing without considering the internal source code [8], [9], [10], [11], [13], [14], [24] or requires manually annotated rules to derive input preconditions [11] or path conditions [12]. In contrast, our technique leverages execution path feedback during input generation and does not require manual efforts.

Model-level fuzzing. Model-level DL library or compiler testing aims to detect integration bugs of DL operators by testing computation graphs. Earlier studies [7], [29], [30] directly use or mutate existing trained models. Recent research [28], [31], [32], [33] generates computational graphs for fuzzing to cover more DL operators. These newer approaches handle constraints between computation nodes by either inserting reshape operators [28], using handcrafted specification [31], leveraging program synthesis [32], or generating DL models with diverse layer API calls [33]. Our work performs API-level fuzzing and focuses on fuzzing a single API in isolation to thoroughly test and explore diverse API behaviors.

B. Concolic Testing

Concolic testing has been extensively studied for decades [21], [22], [34]. Our work relates to those techniques that are aimed at addressing the overhead from heavyweight instrumentation and complex constraint-solving challenges associated with concolic testing. One well-established strategy for addressing this challenge is hybrid fuzzing, which combines concolic testing with greybox fuzzing [27], [35], [36], [37]. This approach mitigates the overhead issue by reducing the computational cost through the alternating use of concolic testing and greybox fuzzing. In contrast, PATHFINDER addresses this overhead issue in a fundamentally different way. First, PATHFINDER does not require symbolic execution to obtain path conditions, thus it eliminates the need for heavyweight instrumentation. Instead, PATHFINDER requires as light instrumentation as greybox level to track executed branches. Additionally, PATHFINDER mitigates the constraint-solving overhead by using approximate path conditions rather than exact ones. This trade-off between precision and efficiency proves to be effective, as demonstrated by our experimental results in fuzzing deep learning libraries.

IX. Conclusion

This paper presents a novel lightweight concolic testing technique for deep learning (DL) libraries. Unlike previous techniques that perform blackbox fuzzing without considering the internal structure of the program under test during input generation, our approach explores diverse execution paths by inferring approximate path conditions. While traditional concolic testing requires heavy overhead for maintaining and interpreting symbolic expressions along the execution path, our technique quickly synthesizes branch conditions based on the observed behaviors of program executions. The evaluation of our tool, PATHFINDER on PyTorch and TensorFlow, shows that PATHFINDER significantly outperforms existing API-level DL fuzzing techniques by achieving higher branch coverage and detecting more unique bugs. PATHFINDER found 61 new crash bugs of which 59 were confirmed by developers.

Acknowledgement

We thank the anonymous reviewers for their invaluable feedback. This work is supported by the National Research Foundation of Korea (NRF) grants funded by the Korea government (MSIT) (No. RS-2022-NR069867, RS-2022-NR066503, RS-2021-NR064479, RS-2021-NR060080) and the Institute for Information & Communications Technology Planning & Evaluation (IITP) grants funded by the Korea government (MSIT) (No. RS-2024-00337414, RS-2024-00437306, RS-2020-II201336).

References

[1] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, 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 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[2] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[3] R. Zhang, W. Xiao, H. Zhang, Y. Liu, H. Lin, and M. Yang, "An empirical study on program failures of deep learning jobs," in *Proceedings of the 42nd International Conference on Software Engineering (ICSE '20)*. IEEE/ACM, July 2020.

[4] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan, "A comprehensive study on deep learning bug characteristics," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 510–520.

[5] Y. Zhang, Y. Chen, S.-C. Cheung, Y. Xiong, and L. Zhang, "An empirical study on tensorflow program bugs," in *Proceedings of the 2020 International Symposium on Software Testing and Analysis*, ser. ISSTA 2018, 07 2018, pp. 129–140.

[6] N. Humbatova, G. Jahangirova, G. Bavota, V. Riccio, A. Stocco, and P. Tonella, "Taxonomy of real faults in deep learning systems," in *Proceedings of 42nd International Conference on Software Engineering*, ser. ICSE '20. ACM, 2020.

[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 *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19. Piscataway, NJ, USA: IEEE Press, 2019, pp. 1027–1038.

[8] A. Wei, Y. Deng, C. Yang, and L. Zhang, "Free lunch for testing: fuzzing deep-learning libraries from open source," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 995–1007. [Online]. Available: https://doi.org/10.1145/3510003.3510041

[9] 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

[10] 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 IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3623343

[11] 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 *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 176–188. [Online]. Available: https://doi.org/10.1145/3533767.3534220

[12] 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 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 690–702. [Online]. Available: https://doi.org/10.1145/3597926.3598088

[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] 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 *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 7393–7410. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/deng-zizhuang

[15] 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

[16] "libfuzzer – a library for coverage-guided fuzz testing." 2015. [Online]. Available: http://llvm.org/docs/LibFuzzer.html

[17] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 213–223. [Online]. Available: http://doi.acm.org/10.1145/1065010.1065036

[18] K. Sen, D. Marinov, and G. Agha, "Cute: A concolic unit testing engine for c," ser. ESEC/FSE-13. New York, NY, USA: ACM, 2005, pp. 263–272. [Online]. Available: http://doi.acm.org/10.1145/1081706.1081750

[19] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. USA: USENIX Association, 2008, p. 209–224.

[20] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," in *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '08. USA: IEEE Computer Society, 2008, p. 443–446. [Online]. Available: https://doi.org/10.1109/ASE.2008.69

[21] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," vol. 56, no. 2. New York, NY, USA: Association for Computing Machinery, feb 2013, p. 82–90. [Online]. Available: https://doi.org/10.1145/2408776.2408795

[22] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, may 2018. [Online]. Available: https://doi.org/10.1145/3182657

[23] W. Lee, "Combining the top-down propagation and bottom-up enumeration for inductive program synthesis," vol. 5, no. POPL. New York, NY, USA: Association for Computing Machinery, jan 2021. [Online]. Available: https://doi.org/10.1145/3434335

[24] 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

[25] E. W. Myers, "An o(nd) difference algorithm and its variations," *Algorithmica*, vol. 1, pp. 251–266, 1986.

[26] L. De Moura and N. Bjørner, "Z3: an efficient smt solver," in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008, p. 337–340.

[27] J. Choi, J. Jang, C. Han, and S. K. Cha, "Grey-box concolic testing on binary code," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 736–747.

[28] 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

[29] Z. Wang, M. Yan, J. Chen, S. Liu, and D. Zhang, "Deep learning library testing via effective model generation," in *Proceedings of the 2020 28th ACM Joint Meeting on European Software Engineering Conference and*

Symposium on the Foundations of Software Engineering, ser. ESEC/FSE 2020, 2020.

[30] 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

[31] 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 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 530–543. [Online]. Available: https://doi.org/10.1145/3575693.3575707

[32] 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

[33] M. Li, J. Cao, Y. Tian, T. O. Li, M. Wen, and S.-C. Cheung, "Comet: Coverage-guided model generation for deep learning library testing," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 5, jul 2023. [Online]. Available: https://doi.org/10.1145/3583566

[34] X. Zhu, S. Wen, S. Camtepe, and Y. Xiang, "Fuzzing: A survey for roadmap," *ACM Comput. Surv.*, vol. 54, no. 11s, sep 2022. [Online]. Available: https://doi.org/10.1145/3512345

[35] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," 01 2016.

[36] L. Borzacchiello, E. Coppa, and C. Demetrescu, "Fuzzolic: Mixing fuzzing and concolic execution," *Comput. Secur.*, vol. 108, no. C, sep 2021. [Online]. Available: https://doi.org/10.1016/j.cose.2021.102368

[37] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM : A practical concolic execution engine tailored for hybrid fuzzing," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 745–761. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/yun