Coverage-Guided Tensor Compiler Fuzzing with Joint IR-Pass Mutation

JIAWEI LIU, University of Illinois at Urbana-Champaign, USA
YUXIANG WEI*, Tongji University, China
SEN YANG*, Fudan University, China
YINLIN DENG, University of Illinois at Urbana-Champaign, USA
LINGMING ZHANG, University of Illinois at Urbana-Champaign, USA

In the past decade, Deep Learning (DL) systems have been widely deployed in various application domains to facilitate our daily life, e.g., natural language processing, healthcare, activity recognition, and autonomous driving. Meanwhile, it is extremely challenging to ensure the correctness of DL systems (e.g., due to their intrinsic nondeterminism), and bugs in DL systems can cause serious consequences and may even threaten human lives. In the literature, researchers have explored various techniques to test, analyze, and verify DL models, since their quality directly affects the corresponding system behaviors. Recently, researchers have also proposed novel techniques for testing the underlying operator-level DL libraries (such as TensorFlow and PyTorch), which provide general binary implementations for each high-level DL operator and are the foundation for running DL models on different hardware platforms. However, there is still limited work targeting the reliability of the emerging tensor compilers (also known as DL compilers), which aim to automatically compile high-level tensor computation graphs directly into high-performance binaries for better efficiency, portability, and scalability than traditional operator-level libraries. Therefore, in this paper, we target the important problem of tensor compiler testing, and have proposed Tzer, a practical fuzzing technique for the widely used TVM tensor compiler. Tzer focuses on mutating the low-level Intermediate Representation (IR) for TVM due to the limited mutation space for the high-level IR. More specifically, Tzer leverages both general-purpose and tensor-compiler-specific mutators guided by coverage feedback for diverse and evolutionary IR mutation; furthermore, since tensor compilers provide various *passes* (i.e., transformations) for IR optimization, Tzer also performs pass mutation in tandem with IR mutation for more effective fuzzing. Our experimental results show that Tzer substantially outperforms existing fuzzing techniques on tensor compiler testing, with 75% higher coverage and 50% more valuable tests than the 2nd-best technique. Also, different components of Tzer have been validated via ablation study. To date, Tzer has detected 49 previously unknown bugs for TVM, with 37 bugs confirmed and 25 bugs fixed (PR merged).

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: Fuzzing, Compiler Testing, Machine Learning Systems

ACM Reference Format:
Jiawei Liu, Yuxiang Wei, Sen Yang, Yinlin Deng, and Lingming Zhang. 2022. Coverage-Guided Tensor Compiler Fuzzing with Joint IR-Pass Mutation. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 73 (April 2022), 26 pages.
https://doi.org/10.1145/3527317

*The work was done during a remote summer internship at University of Illinois.

Authors' addresses: Jiawei Liu, University of Illinois at Urbana-Champaign, USA, jiawei6@illinois.edu; Yuxiang Wei, Tongji University, China, nolest@tongji.edu.cn; Sen Yang, Fudan University, China, syang15@fudan.edu.cn; Yinlin Deng, University of Illinois at Urbana-Champaign, USA, yinlind2@illinois.edu; Lingming Zhang, University of Illinois at Urbana-Champaign, USA, lingming@illinois.edu.

2475-1421/2022/4-ART73
https://doi.org/10.1145/3527317

1 INTRODUCTION

With the recent advance of deep learning (DL), DL systems have been pervasively deployed in various application domains to facilitate our daily life, including natural language processing [Devlin et al. 2019; Vaswani et al. 2017; Young et al. 2018], healthcare [Esteva et al. 2019; Miotto et al. 2018], activity recognition [Cao et al. 2019; Guo et al. 2021; Kreiss et al. 2019], and autonomous driving [Grigorescu et al. 2020; Rao and Frtunikj 2018]. Meanwhile, it is extremely challenging to ensure the correctness of DL systems (e.g., due to their intrinsic nondeterminism), and any bug in such decision-making systems can potentially bring serious consequences or accidents (e.g., the life-threatening autonomous-driving failures [Garcia et al. 2020]).

To date, a large body of prior work has been dedicated to testing, analyzing, and verifying DL models since their quality directly affects the behaviors of DL systems. For example, various techniques have been designed to generate adversarial or edge-case model inputs for testing DL models, including DeepXplore [Pei et al. 2019], DeepTest [Tian et al. 2018], DeepRoad [Zhang et al. 2018], TensorFuzz [Odena et al. 2019], and DeepBillboard [Zhou et al. 2020]. In recent years, in addition to the algorithmic/model aspect, researchers also realized the importance of ensuring the correctness of the underlying DL infrastructure supports, and have proposed novel techniques [Pham et al. 2019; Wang et al. 2020] specifically targeting operator-level DL libraries, such as TensorFlow [Abadi et al. 2016] and PyTorch [Paszke et al. 2019]. Meanwhile, computation-intensive DL models are being developed everywhere nowadays; early operator-level libraries, which usually only provide a fixed binary for a limited number of platforms, are hardly generalizable and scalable. Therefore, DL engineers and researchers have been building an ultimate solution, tensor compilers [Chen et al. 2018; Lattner et al. 2020; Ragan-Kelley et al. 2013; Rotem et al. 2018] (also known as DL compilers), to essentially tackle the challenges in performance, portability, and flexibility. However, to our best knowledge, there is limited work specifically targeting the reliability of the emerging tensor compilers.

Ensuring the correctness and reliability of tensor compilers is essential for the rise of compilation-based DL infrastructure. Nonetheless, the complicated software stack of tensor compilers makes it non-trivial for writing hand-crafted unit tests. For example, in TVM [Chen et al. 2018] (one of the biggest and most widely used tensor compiler projects), there are over 117k lines of Python code specifically targeting unit testing! Designing automated testing techniques for tensor compilers is important but also quite challenging. First, the compiler stack is *deep*, meaning that an input model needs to be compiled through various phases (including numerous parsing, lowering, and optimization passes) to produce the final target code. Second, the compiler stack is *wide*, meaning that there are innumerable possibilities for composing a single intermediate representation (IR) file or an optimization sequence, let alone their combinations if taking various targets and execution backends into account.

Although some existing fuzzing techniques can potentially be adopted for testing tensor compilers, they are not able to handle the complex compiler infrastructure well. For example, general-purpose binary fuzzers [Serebryany 2016; Zalewski 2018] can hardly generate syntactically- and semantically-valid inputs, wasting the majority of time fuzzing the lexical parsing components. Prior operator-level DL-library testing techniques [Wang et al. 2020] systematically mutate on the input model seeds to generate diverse model architectures, and can potentially be generalized to most DL infrastructures; however, they are not tailored for tensor compiler testing as they do not consider triggering different optimizations and are also too coarse-grained to generate light-weight yet valuable inputs (as demonstrated by our experimental results in § 5.1). To our best knowledge, the only existing work specifically targeting tensor compiler fuzzing, TVMFuzz [Pankratz 2020], employs a generation-based approach to automatically generate arbitrary low-level IRs for fuzzing TVM. However, it suffers from the common limitations of generation-based fuzzing techniques [Holler et al. 2012; Yang et al. 2011], e.g., it is challenging to simulate realistic programs to cover deep code paths and the fuzzing process lacks

valid guidance; also, it fails to consider the rich search space of possible optimization pass sequences for tensor compilers. As a result, it could only find out very shallow front-end bugs and its coverage growth converges at an early stage (as also confirmed by our experimental results).

In this paper, we focus on practical tensor compiler fuzzing and have made the following design choices. First, we target low-level IR mutation due to the coarse-grained and limited mutation space for high-level IR mutation [Wang et al. 2020]. Second, we propose the first coverage-guided fuzzing approach for testing tensor compilers, as coverage feedback has been demonstrated to be powerful for exploring deep code paths efficiently in general [Li et al. 2018]. Following traditional coverage-guided fuzzers [Serebryany 2016; Zalewski 2018], in each iteration, we randomly choose an IR file from a seed pool for mutation and add the newly mutated IR file into the pool only when it triggers new coverage. Meanwhile, instead of relying on the bit-level mutators widely adopted in traditional fuzzers, we develop a set of general-purpose and tensor-compiler-specific mutators for more targeted and effective IR mutation. Third, since a large number of optimization passes can form a pass sequence and potentially be applied to the same IR file to trigger different compiler behaviors, we further build a novel coverage-guided fuzzing strategy to perform joint mutations of both IR and optimization passes for more exhaustive tensor compiler testing. Although our design is general for different tensor compilers, in this paper, we mainly focus on the TVM compiler and have implemented a practical TVM fuzzing technique named Tzer. To evaluate the effectiveness of Tzer, we have performed an extensive study to compare Tzer against LibFuzzer [Serebryany 2016] (a state-of-the-art general-purpose fuzzer), LEMON [Wang et al. 2020] (a state-of-the-art high-level IR fuzzer for DL libraries), and TVMFuzz [Pankratz 2020] (the only existing low-level IR fuzzer for TVM). Furthermore, we have rigorously evaluated the importance and necessity for all the design choices of Tzer. In summary, the primary contributions of this work go as follows:

- **Novelty**: This paper presents the first coverage-guided fuzzing technique specifically targeting tensor compilers. More specifically, we have designed various general-purpose and tensor-compiler-specific mutators as well as the joint mutation of both IR and optimization passes for effective tensor compiler fuzzing.
- **Implementation**: We have implemented the proposed technique as a practical fuzzer (named Tzer) for the TVM compiler. Tzer is mainly implemented by over 8.7k lines of Python code together with ~150 lines of C++ code for extending the LLVM Coverage Sanitizer. Tzer has been open-sourced at https://github.com/ise-uiuc/tzer.
- **Study**: We have performed an extensive study to compare Tzer against existing fuzzers for testing TVM, and have also rigorously validated the contribution of each component of Tzer. The experimental results show that Tzer is able to substantially outperform state-of-the-art fuzzers with 75% higher coverage and 50% more valuable tests compared with the 2nd-best fuzzer. Furthermore, different components of Tzer all contribute to its final effectiveness. To date, among 49 unique new bugs[1] found by Tzer, 37 bugs have been confirmed and 25 of them have been fixed and merged to the main branch of TVM.

2 BACKGROUND AND RELATED WORK

2.1 Tensor Compilers

The computation of deep learning models can be logically described in the dataflow model [Wongsuphasawat et al. 2017], which is commonly called the *computation graph* [Jia et al. 2019]. A computation graph consists of a number of operators (e.g., convolution, max pooling, and many other tensor operations), each of which transforms one or multiple input tensors (i.e., multi-dimensional arrays) into a series of output tensors. Given the computation graph description, there are mainly two approaches for

[1]We count the number of bugs by unique root fixes (see § 4.4).

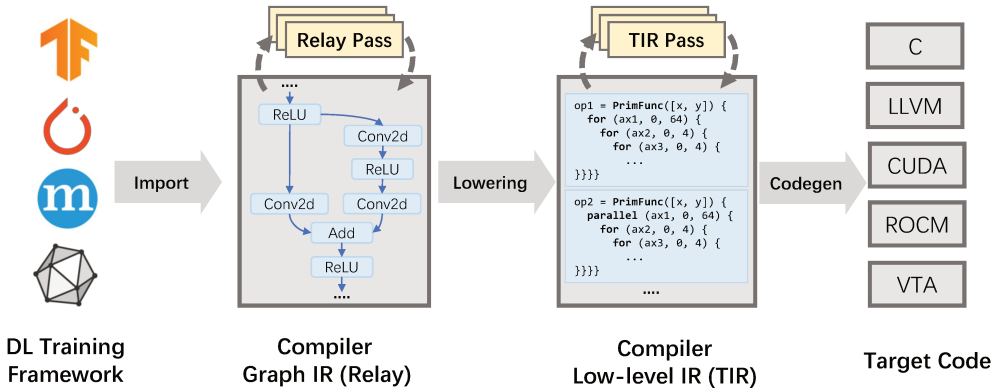

Fig. 1. Compilation Flow of TVM.

existing DL software to compute it. Previously, for fast software delivery, ML engineers implemented various operator-level DL libraries, such as TensorFlow [Abadi et al. 2016] and PyTorch [Paszke et al. 2019], whose operators are implemented with fixed and hand-optimized kernel functions. However, hand-crafted optimization is time-consuming in the long run and a fixed binary cannot meet the ultimate performance requirements for all hardware vendors. Therefore, to fundamentally resolve those challenges, recently DL infrastructures have been focusing on developing tensor compilers [Chen et al. 2018; Google 2016; Intel 2017; Jin et al. 2020; Rotem et al. 2018; Tillet et al. 2019; Zhao et al. 2021] to automatically generate best-in-class target code for different vendors or even architectures.

Figure 1 illustrates the compilation flow of TVM [Chen et al. 2018], one of the most widely-used and advanced tensor compilers (other tensor compilers including XLA [Google 2016] and Glow [Rotem et al. 2018] also follow such logical flow). First, tensor compilers will transform 3rd-party model files into their own graph representation (i.e., Relay IR in TVM). Furthermore, a sequence of optimizations (known as *passes* or *transformations*) is applied for either high-level graph IRs and low-level Tensor IRs (TIR). Within a pass sequence, each pass iteratively transforms an IR to a new IR to either optimize the computation or propagate valuable information for upcoming optimizations. Once the low-level IR is ultimately optimized, the code generation component will produce corresponding binaries for different targets (i.e., NVIDIA GPU, X86 CPU, etc.)

Existing work on DL-library testing [Pham et al. 2019; Wang et al. 2020; Wei et al. 2022] mainly focuses on testing at the graph-operator level. For example, while CRADLE [Pham et al. 2019] and LEMON [Wang et al. 2020] focus on leveraging/mutating existing DL models for differential testing of DL libraries, FreeFuzz [Wei et al. 2022] directly performs fuzz testing at the DL API level via mining API inputs from open source (including code snippets from library documentation, library tests, and DL models in the wild), and has reported state-of-the-art results for DL library testing. Contrastingly, for tensor compilers, we target the low-level representation since there are many limitations if the input files are simply constructed via such graph-level abstraction. First, low-level IRs are closer to code generation and optimization which can guide the fuzzers to find deeper compiler bugs. Second, there is a limited search space for graph-level construction since deep learning operators are too coarse-grained and it suffers from various shape constraints. Furthermore, graph-level representation can be lowered to concrete low-level IR but not vice versa. In this work, we have empirically compared our Tzer technique that operates on the low-level IRs with state-of-the-art DL-library fuzzer, LEMON [Wang et al. 2020], which performs graph-level model mutation. The evaluation results also confirm that LEMON generates 7.7x less valuable tests (i.e., the tests that are compilable and can trigger new compiler coverage) compared with Tzer.

2.2 Fuzzing

Fuzzing [Böhme et al. 2017; Fioraldi et al. 2020; Lemieux and Sen 2018; Serebryany 2016; Wu et al. 2022; Zalewski 2018; Zhao et al. 2022], known as an advanced automatic testing technique, has been widely employed to efficiently detect software bugs in the wild. The key features of fuzzing, is the extreme 1) *efficiency*: no heavy-weight analysis is required, and 2) *simplicity*: fuzzers are mostly general-purpose and could be as easily employed as compiling a program and then executing it.

The big idea of fuzzing, is to generate randomized inputs in sharp and explore unexpected behaviors (e.g., crashes) of the program under test. One of the most effective techniques of fuzzing is called the coverage-guided fuzzing (CGF), which is a *mutation-based* approach that leverages coverage feedback to focus on test inputs (known as *seeds*) that have achieved new coverage, instead of doing so in a randomized fashion.

The idea of CGF has led to many existing general-purpose binary fuzzers both in industry and in research [Böhme et al. 2017; Fioraldi et al. 2020; Lemieux and Sen 2018; Serebryany 2016; Zalewski 2018]. AFL [Zalewski 2018] is one of the pioneers among CGF tools that have found numerous vulnerabilities in diverse applications. The development of AFL has inspired many further enhancements and extensions. AFLFast [Böhme et al. 2017] further leverages the Markov chain to model CGF as a systematic exploration of its state space and develops a set of power schedules and search strategies to focus on low-frequency paths. FairFuzz [Lemieux and Sen 2018], which outperforms AFLFast in its evaluation, prioritizes seeds that hit rare branches, instead of rare paths, and develops a mutation mask algorithm to bias mutation towards producing inputs that hit such rare branches. AFL++ [Fioraldi et al. 2020] further incorporates state-of-the-art fuzzing research ideas into one useful tool, which is prospective to be a new baseline tool for future research in Fuzzing. LibFuzzer [Serebryany 2016] has been widely recognized as one of the most representative coverage-guided fuzzers that builds in-process fuzzing loop and powerful evolutionary fuzzing engine with its integration with the LLVM infrastructure [Lattner 2002]. It has been under active development and keeping adopting the most recent and influential research ideas [Böhme et al. 2020].

In addition to general-purpose fuzzers, CGF has also inspired many domain-specific fuzzers. Die [Park et al. 2020], an aspect-preserving evolutionary fuzzing technique for JavaScript, has been shown to outperform state-of-the-art JavaScript fuzzers in terms of both bug discovery and valid test input generation. Squirrel [Zhong et al. 2020] is a database management system (DBMS) fuzzer that takes language validity into consideration during fuzzing, which has found numerous bugs in DBMSs including SQLite, MySQL, PostgreSQL, and MariaDB. FuzzChick [Lampropoulos et al. 2019], an extension of QuickChick [Dénès et al. 2014], incorporates coverage guidance to perform property-based testing for Coq programs, and has been shown to perform far better than the vanilla QuickChick with the help of coverage guidance.

The existing general-purpose fuzzers cannot be simply applied here for tensor compilers like TVM because tensor compilers require structural IRs in specific form as input, which does not have a direct correspondence to the binary stream. Furthermore, many traditional compiler fuzzing techniques [Le et al. 2014; Yang et al. 2011; Zhang et al. 2017], though also theoretically general and applicable, are insufficient for tensor compiler fuzzing as they are not tailored for such purposes. For instance, the well-known EMI [Le et al. 2014] is general for any compilers supporting control flows. However, it is not suitable for DL computation as most existing DL models are static graphs (i.e., no control flows) mainly except for some RNN models. In addition, in TVM the de facto compilation mode (i.e., the "graph" mode) requires constant input tensor shape so that any control flows related to shape sizes can be statically inferred to allow maximum optimization (e.g., unrolling loops in an optimal way), making it unsuitable for applying EMI. To date, there are very few domain-specific fuzzers for tensor compilers, with TVMFuzz [Pankratz 2020] being the only existing fuzzer specifically targeting

TVM to our knowledge. Therefore, this paper aims to build a practical fuzzing technique specifically targeting modern tensor compilers.

3 APPROACH

In this section, we present the detailed design of TZER, a practical tensor compiler fuzzer via coverage-guided joint IR-Pass mutation. Figure 2 illustrates the overview of TZER. As shown in the figure, like traditional coverage-guided fuzzing work [Li et al. 2018], TZER maintains a seed pool to store interesting seeds (i.e., the test inputs that can trigger new coverage) for further mutations. Different from prior work that mainly maintains the input files within the seed pool, TZER maintains two dimensions of information in the seed pool (i.e., both IR files and their corresponding optimization pass sequences) for effective joint IR-pass mutation.

During the fuzzing process, for each pair of IR and pass sequence from the seed pool, TZER will apply the corresponding mutation strategies to generate a new input pair in each iteration. For example, TZER applies both general-purpose and tensor-compiler-specific mutators on IR files to generate new IR files, and applies pass mutation to randomly generate a new pass sequence. Then, for each newly generated IR-pass pair, TZER leverages the tensor compiler under test (i.e., TVM in this work) to compile IR with the corresponding pass sequence and collect the compiler coverage information. Any input pairs that violate the test oracles are reported, while any input pairs that can help trigger new compiler coverage are further fed back to the seed pool for generating more valuable inputs. In this way, the generated inputs can cover more and more code for tensor compilers, and can detect more and more potential bugs. The fuzzing loop will terminate until the allowed time/resource budget runs out.

In the remainder of this section, we will first present the detailed algorithm design for our fuzzing loop (§ 3.1). Then, we will present the details for our general-purpose mutators (§ 3.2) and tensor-compiler-specific mutators (§ 3.3). Finally, we will briefly discuss the test oracle information used in this work (§ 3.4).

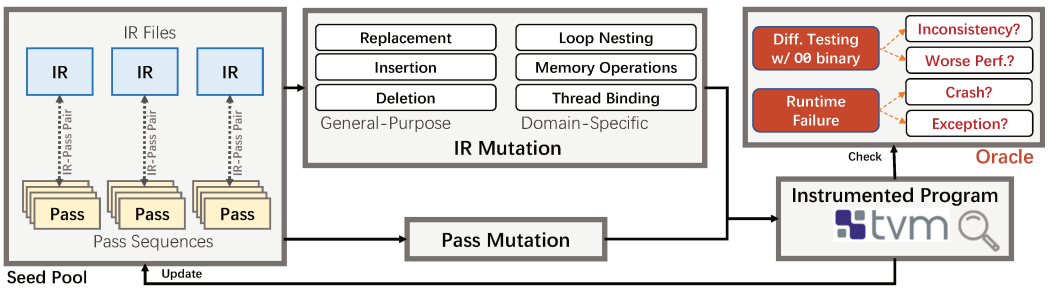

Fig. 2. Overview of TZER

3.1 Fuzzing Loop

Algorithm 1 presents the detailed design of our main TZER fuzzing loop. The algorithm only takes three inputs, including the initial seed pool (S_0), the time budget (T), and the parameter for controlling the interleaving of IR and pass mutations (N_{max}). Different from all prior work on evolutionary coverage-guided fuzzing [Serebryany 2016; Zalewski 2018], the seed pool of TZER maintains two dimensions of information for effective tensor-compiler fuzzing, i.e., both the IR files and their corresponding pass sequences. Thus, we can denote each input for TZER as a pair $\langle F,P\rangle$, where F represents an IR file while P represents the corresponding pass sequence for the IR. In the algorithm, we further extend $\langle F,P\rangle$ into $\langle F,P,N\rangle$ to additionally consider the interleaving control N for the join IR-pass mutation. With the interleaving control N, for each seed input IR file F, TZER can 1) keep

mutating F with P if such mutations were rewarding, and 2) also occasionally (controlled by N) seek a better P' to pair with F when the current P get stuck in local minima.

The main algorithm of Tzer is similar to traditional evolutionary fuzzers, except for the additional code logic added to handle the additional pass mutation (highlighted in colored boxes). Basically, Tzer first initializes the seed pool with pairs of IR files and pass sequences, as well as setting N=0 for all pairs (Line 2). For example, in this work, the initial seed pool consists of all possible model architectures in the TVM model zoo [Community 2020] with randomly generated pass sequences. The coverage achieved by the initial seed inputs is also collected to evaluate newly generated inputs (Line 3). Then, Tzer will go through the main loop for generating new inputs (Lines 4-26).

In each iteration, Tzer will randomly fetch an input tuple from the seed pool. If the current $\langle F,P\rangle$ pair cannot trigger any new coverage during the past N_{max} consecutive IR mutations (Line 6), Tzer will try to mutate the pass sequence P into another random sequence P' in the hope that P' will bring this input pair to a better state for further mutations (Line 7). The coverage and error information will be recorded when compiling the input pair $\langle F,P'\rangle$ with the compiler under test (Line 8). In case of any error, the input pair will be reported to the developers. If P' does help trigger new coverage, the total coverage information will be updated (Line 12); the input pair $\langle F,P,N\rangle$ in the seed pool will also be updated to $\langle F,P',0\rangle$ since it is more promising to go with P' in future runs on mutating F (Lines 13). If

Algorithm 1: Tzer Fuzzing Loop

```
1  Function Fuzz(set of initial seeds S_0, time budget T, pass mutation frequency control N_max):
2      S ← S_0
3      C_total ← ⋃_{i∈S_0} Coverage(i)
4      while within time budget T do
5          ⟨F,P,N⟩ ← Select(S)
6          if N = N_max then
7              P' ← MutatePass(P)
8              err,cov ← ExecuteTVM(F,P')
9              if ∃err then
10                 Report(F,P')
11             else if cov ⊈ C_total then
12                 C_total ← C_total ∪ cov
13                 Update(S,⟨F,P',0⟩)
14             else
15                 Update(S,⟨F,P,0⟩)
16             Continue
17         F' ← MutateIR(F)
18         err,cov ← ExecuteTVM(F',P)
19         if ∃err then
20             Report(F',P)
21         else if cov ⊈ C_total then
22             S ← S ∪ ⟨F',P,0⟩
23             C_total ← C_total ∪ cov
24             Update(S,⟨F,P,0⟩)
25         else
26             Update(S,⟨F,P,N+1⟩)
```

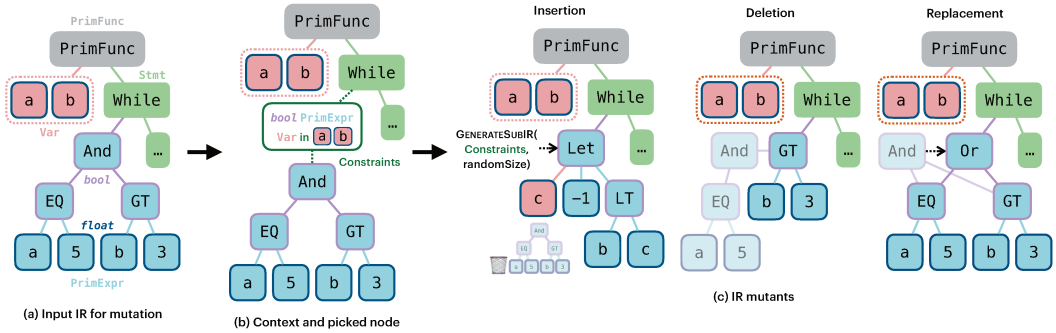

Fig. 3. The process of IR mutation. Node types are differentiated using background colors, while expression types are differentiated using border colors. Each label around the node denotes a node type or an expression type. Each label on the node denotes a constructor or a primitive. The `VarInjection` constructor in Table 1 is illustrated by switching the background color of the `Var` node.

P' does not help trigger new coverage, Tzer simply clears the interleaving control counter to 0 to allow more file mutations with the current P (Line 15). This indicates that we do not perform consecutive pass mutations for any seed input regardless of the coverage outcome. The reason is that mutating pass sequences is not as rewarding as mutating IR files in general, and we only need *occasional* pass mutation (controlled via N) to guide the evolutionary process to more promising states to avoid local minima.

On the other hand, if the fetched input tuple has not failed to trigger new coverage for N_{max} consecutive IR mutations, Tzer will go ahead to further mutate the IR file following a very similar process to traditional fuzzers. Tzer first mutates the IR F into F' by selecting one mutator among the mutator pool (including 3 general-purpose and 3 domain-specific mutators), and then collects the result information for compiling the pair $\langle F', P\rangle$ (Lines 17 and 18). In case of any error, the input pair will be reported. If F' does help trigger new coverage, the new IR file with the current P will be inserted into the seed pool for future runs (Line 22). The total coverage information will also be updated (Line 23). Different from prior fuzzers, Tzer also needs to update the original seed pair to $\langle F, P, 0\rangle$ since it helped trigger new coverage (Line 24); also, if F' did not help achieve new coverage, the original seed pair will be updated to $\langle F, P, N+1\rangle$ to record the current attempt that failed to trigger new coverage (Line 26).

Theoretically, some specific $\langle F, P\rangle$ might fail due to 1) lack of pass dependency, or 2) pass/IR incompatibility, resulting in waste of time for compiling invalid $\langle F, P\rangle$. Executing too many invalid compilations will make fuzzing process less efficient. The evolutionary joint IR-Pass mutation (Algorithm 1) can easily avoid such frequent invalid compilation by design. As is shown in Line 22, only valid $\langle F, P\rangle$ with new coverage will be added into the seed pool $\mathcal{S}$, whereas the invalid and ineffective ones will be ignored to keep seed pool filled with compilable samples during evolutionary fuzzing process.

In this way, after being launched, the algorithm can then continuously generate valuable IR and pass sequence pairs for triggering tensor-compiler bugs.

3.2 General-Purpose Mutation

Following prior work on fuzzing programming languages [Holler et al. 2012; Lampropoulos et al. 2019; Zhong et al. 2020], we design a general-purpose IR mutation approach. In addition, program analysis techniques are integrated into mutation to ensure syntax correctness and to mitigate semantic errors. This is because, to dig high-quality bugs in the code generation and optimization phases of a compiler, the produced IRs should be able to pass standard pre-condition checks (e.g., syntax checks and semantics checks). In the remainder of this section, we first introduce our definition of the low-level Tensor IR (TIR) of TVM and then elaborate on the the mutation details.

We first discuss the abstract syntax tree (AST) of TIR since it is the entry point of TVM's compilation. Figure 3a depicts a simplified TIR AST sample. The AST tree contains different types of nodes, with the root node representing the input IR to the compiler. As is shown Figure 3a, the type of the root node is **`PrimFunc`** which stands for the basic function type in TIR. The `While` node is of type **`Stmt`** while the `EQ` and `GT` nodes are of type **`PrimExpr`**. The corpus of all node types can be defined as

$$\textbf{NodeTypes} = \{NodeType_1, NodeType_2, ..., NodeType_n\}. \tag{1}$$

We assume these types are disjoint (e.g., no subtype relation), but our implementation leverages the concept of *constructor* to simulate the original subtype relations. For each type, there could be multiple *constructors*, which are functions/operators from one or more node types to one return node type, generally having the signature

$$(NodeType_{i_1}, NodeType_{i_2}, ..., NodeType_{i_p}) \rightarrow NodeType_{i_r}. \tag{2}$$

Some types also have *primitives*, which are values that cannot be broken down into subparts (i.e., leaf nodes).

Table 1 shows a detailed list of common TIR AST node types, constructors, and primitives. Note that `VarInjection`, one *constructor* of **`PrimExpr`**, is added by us to switch the variable from type **`Var`** to **`PrimExpr`** without changing its internal value. This is required because **`Var`** is a *subtype* of **`PrimExpr`** in the implementation of TIR by TVM, which means each **`Var`** is implicitly a **`PrimExpr`**, but in our definition, we assume no subtype relation. By using injective constructors, this could be easily expressed.

Each AST node can be recursively defined as either a primitive (leaf node) or an application of some constructor to other nodes (e.g., branch node). For simplicity, in our implementation of mutation approaches, some trivial branch nodes are treated as leaf nodes, including `Var`, `IntImm`, `FloatImm`, etc. As an example, the root node of the input IR in Figure 3a can be formally defined as

$$\texttt{PrimFunc([a,b],While(And(EQ(a,5),GT(b,3)),...))}:\textbf{PrimFunc}. \tag{3}$$

The first step of our mutation approach is to randomly pick out one of the AST nodes of the given IR and regard it as a hole, which can then be filled up to produce an IR mutant. We call an IR with a hole at some position a *context*. For instance, in Figure 3b, we pick out the `And` node as a hole (denoted by □) so that the corresponding context is

$$\texttt{PrimFunc([a,b],While(□,...))}: Context. \tag{4}$$

Based on the context, we can derive the *constraints* to be satisfied (e.g., accessible variables of the hole) when filling the hole so that the filled IR could be correct. Formally, the constraints are a tuple of necessary information that helps determine the requirements when constructing a sub-expression in the hole. Specifically, for TIR, we consider the following information:

- *Desired AST node type (e.g.,* **`PrimExpr`**, **`Stmt`**, **`Var`***).*
- *Desired expression type (e.g.,* `int32`, `float32`, `bool`*).*
- *Accessible variables under the current scope.*
- *Declared buffers.* TIR uses the notion of "buffer" to store and load data. When we access a buffer, we should ensure it is already declared.
- *A boolean indicating whether the variables need to be bound.* TIR only allows a commented expression to have free variables.

As an example, for the context in Equation 4, the hole represents a condition check for the `While` node. Hence, in order to fill the hole, at least a boolean expression is needed. Also, any variable used should be bound to some binding occurrence (e.g., parameter `a` and `b`). Therefore, the constraints should be

$$(\textbf{PrimExpr}, \texttt{bool}, \texttt{[a,b]}, \texttt{[]}, \texttt{true}): Constraints. \tag{5}$$

Table 1. Example node types, constructors, and primitives of the AST of TIR for Figure 3. Some constructor could be overloaded or have the same name as their node types. The asterisk '*' means a list type (e.g., `Var`* in the two `PrimFunc` constructors means a list of `Var`, which serves as parameters of a function). We also put several auxiliary labels in front of some parameter types to help understand the meaning of the parameter (e.g., "*name*:`String`" in the `Var` constructor signature).

Node Type	Constructors and Primitives
`PrimFunc`. Function type, the input type of the compiler.	`PrimFunc`:(**`Var`***,**`Stmt`**) → **`PrimFunc`** `PrimFunc`:(**`Var`***,**`Stmt`**,**`Buffer`**) → **`PrimFunc`**
`Buffer`. Buffer type, describing the storage of data.	`decl_buffer`:(**`Shape`**,**`DataType`**) → **`Buffer`**
`DataType`. Basic numeric data types for variables and expressions, including integer, boolean, floating point, etc.	`float32`:**`DataType`** `int32`:**`DataType`** `uint32`:**`DataType`** `bool`:**`DataType`**
`Var`. Variable type, used for variable declaration and as expressions.	`Var`:(*name*:**`String`**,**`DataType`**) → **`Var`**
`Stmt`. Statement type, the fundamental building block to form a function. There are sequential statements, control flow statements, etc. Particularly, statements constructed by the `For` constructor are central to many low level optimizations.	`While`:(**`PrimExpr`**,**`Stmt`**) → **`Stmt`** `For`:(**`Var`**,*min*:**`PrimExpr`**,*extent*:**`PrimExpr`**,**`ForKind`**,**`Stmt`**) → **`Stmt`** `IfThenElse`:(**`PrimExpr`**,**`Stmt`**,**`Stmt`**) → **`Stmt`** `LetStmt`:(**`Var`**,**`PrimExpr`**,**`Stmt`**) → **`Stmt`** `SeqStmt`:(**`Stmt`***) → **`Stmt`** `BufferStore`:(**`Buffer`**,**`PrimExpr`**,*indices*:**`PrimExpr`***) → **`Stmt`**
`PrimExpr`. Expression type, the fundamental building block to form a statement. The constructors include basic operators that handle numeric values, buffers, etc. The `Call` constructor is responsible for constructing pre-defined *intrinsics* by TVM. Each **`PrimExpr`** has a corresponding **`DataType`**, either specified explicitly or inferred implicitly.	`VarInjection`:(**`Var`**) → **`PrimExpr`** `And`:(**`PrimExpr`**,**`PrimExpr`**) → **`PrimExpr`** `Or`:(**`PrimExpr`**,**`PrimExpr`**) → **`PrimExpr`** `EQ`:(**`PrimExpr`**,**`PrimExpr`**) → **`PrimExpr`** `GT`:(**`PrimExpr`**,**`PrimExpr`**) → **`PrimExpr`** `LT`:(**`PrimExpr`**,**`PrimExpr`**) → **`PrimExpr`** `Add`:(**`PrimExpr`**,**`PrimExpr`**) → **`PrimExpr`** `Call`:(**`DataType`**,**`Op`**,*args*:**`PrimExpr`***) → **`PrimExpr`** `Let`:(**`Var`**,**`PrimExpr`**,**`PrimExpr`**) → **`PrimExpr`** `BufferLoad`:(**`Buffer`**,*indices*:**`PrimExpr`***) → **`PrimExpr`** `FloatImm`:(**`DataType`**,**`Float`**) → **`PrimExpr`** `IntImm`:(**`DataType`**,**`Int`**) → **`PrimExpr`**

Based on the derived constraints and the picked node, we perform a series of *mutations* using the corresponding *mutator* on the node following the constraints. Formally, each mutator has the signature

$$(\mathit{AnyNodeType}, \mathit{Constraints}) \to \mathit{AnyNodeType}, \tag{6}$$

where *AnyNodeType* is a disjoint union of all possible $\mathit{NodeType} \in \mathbf{NodeTypes}$, i.e.,

$$\mathit{AnyNodeType} = \bigsqcup \mathbf{NodeTypes}. \tag{7}$$

We use a disjoint union here because our mutator is designed to operate on nodes of any node type and different node types should not overlap with each other in our definition.

Basically, we designed the following three general-purpose mutators, namely *Insertion*, *Deletion*, and *Replacement*:

Insertion. Regardless of the input node, Tzer simply returns a new node generated from scratch that satisfies the given constraints. This is done by Tzer's generator, which is inspired by the prior generators in the random testing community [Claessen et al. 2015; Lampropoulos et al. 2017]. The functionality of the generator is to produce IR ingredients/snippets based on the constraints and a *size* parameter which indicates the node size of generated sub-IR, as is described in Figure 3c. In the figure, Tzer generates a new boolean `Let` node of type **`PrimExpr`**, and ensures that all the variable references have their corresponding binding occurrences (e.g., in the node `LT(b,c)`, `b` is introduced by the parameter list, and `c` is introduced by `Let`).

Deletion. Tzer checks the child nodes of the input node, filters out those satisfying the constraints, and randomly returns one of them. For example, in Figure 3c, we perform deletion on the `And` node by returning its right-hand side `GT(b,3)`, the 'greater than' node, which is a boolean expression with all variable references bound.

Replacement. For a primitive node, Tzer simply modifies its value, or returns another primitive based on the constraints. For a node constructed by some constructor, in the simplest case, Tzer randomly selects a constructor to substitute the existing one, in the restriction that after the substitution the node should satisfy the constraints given. More generally, Tzer randomly selects a constructor, trying to use the child nodes of the input node as components to fill the parameter list of the selected constructor; if there are parameters unable to fill, Tzer randomly generates one using the generator. This strategy is inspired by the $\texttt{mutate}_g^T$ constructor of FuzzChick [Lampropoulos et al. 2019] for testing Coq programs except that Tzer considers different constraints. Figure 3c gives the simplest form of replacement, which just replaces the `And` constructor with the `Or` constructor.

3.3 Domain-Specific Mutation

Tensor compilers focus on optimizing domain-specific programs, e.g., programs with dense loops in particular. To optimize those hot spot program structures, existing tensor compilers [Chen et al. 2018; Google 2016; Intel 2017; Jin et al. 2020] leverage the concept of *pass* to optimize the given IR or insert annotations containing valuable information for further optimization. To trigger the complex logic behind those optimization passes, general-purpose mutators, though versatile to handle different types of expressions, are still inefficient and not tailored to the specific domain that tensor compilers are built for.

For domain-specific compiler testing, in addition to the general-purpose mutators, we argue that it is also important to navigate the mutation towards the core components that the compilers specifically target (e.g., loop-oriented optimization, memory allocation, memory latency hiding, and parallelization [Li et al. 2020]). For example, deep and wide nested loops can be optimized with tiling [Park et al. 2003], multi-threading [Smith et al. 2014], and vectorization [Bjørstad et al. 1992] by a series of related passes (e.g., `UnrollLoop` and `LoopPartition`). Those passes have complex optimization rules for different domain-specific code structures (e.g., big loops, large buffer allocation, and thread scheduling) that general-purpose mutators can hardly target. Hence, according to the hot spot program patterns targeted by existing tensor compilers [Chen et al. 2018; Li et al. 2020; Ragan-Kelley et al. 2013; Tillet et al. 2019; Zhao et al. 2021], Tzer specifically designed 3 types of mutators: 1) *loop-nesting* mutator for creating multifarious dense loop structures; 2) *memory-operation* mutator for various memory allocation/store/load patterns at the index level; and 3) *thread-binding* mutator for diversifying the parallel computation flows to generate interesting code patterns that tensor compilers particularly care about.

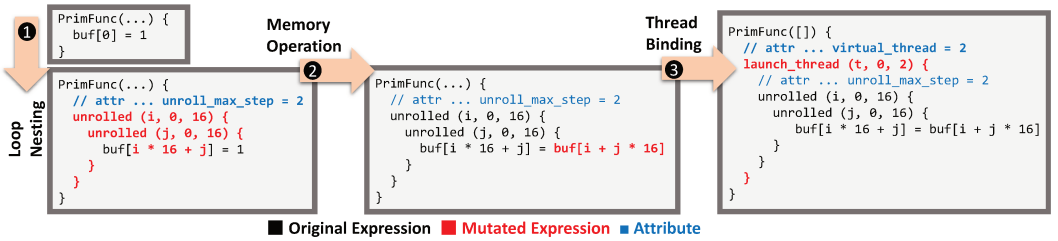

Fig. 4. Example of Domain-Specific IR Mutation

Loop Nesting. Tensor computation usually consists of a large number of nested loops. Even for the simplest element-wise expression, e.g., `C=C+1` with broadcasting, the loop structure of a common image tensor (whose dimensions are [height, width, channels]) will consist of 3 nested loops. To mimic such dense loops, we introduce the loop-nesting mutator to transform IRs with different loop structures.

First, TZER randomly picks an AST node as the innermost loop body. TZER then selects one out of the five TVM loop types (`serial`, `vectorize`, `unroll`, etc.), where each of them represents different control flow semantics. Given the loop type, TZER inserts several loops with either constant or variable loop sizes (constant loops are likely to trigger loop unrolling while variable loops will block such optimization). For example, in step ❶ of Figure 4, 2 nested loops of type `unrolled` are inserted after mutation. Furthermore, according to loop variables under the current context, a random expression will be used to form the indices (`[i*16+j]`). Notably, TVM also annotates loop attributes for concrete optimization in code generation. e.g., `unroll_max_steps`, and further tunes those integer attributes to trigger different optimization paths. Therefore, TZER also mutates those attributes when creating/replacing the target loops.

Memory Operations. Apart from multifarious loop structures, another dimension to increasing the complexity of tensor computation is to introduce various memory operations, including memory store/load and allocation.

TZER's memory-operation mutator mimics complex memory patterns by inserting memory operations into existing IRs. Given a randomly selected node , TZER first analyzes accessible memory buffers (represented with pointers) under the current scope. Next, TZER randomly constructs a memory operation (i.e., a sub-expression) and inserts it into the target AST node. As is shown in Figure 4 (step ❷), TZER inserts a sub-expression (i.e., `... = buf[i+j*16]`) to original IR so that a new memory access is created and the dataflow related to `buf` is changed.

Thread Binding. One thing that differentiates traditional compilers and tensor compilers is that tensor compilers leverage multiple threads (either CPU threads or threads of parallel hardware like NVIDIA GPU) to automatically parallelize the program. The thread scheduling, however, could have many different settings, as operations could be executed by different thread groups at different stages (manipulated by attributes, e.g., thread numbers and thread tags).

To explore the impact of different thread scheduling patterns, TZER creates various thread-binding patterns and leverages them to mutate the multi-thread planning of given IRs. Precisely, as is shown in Figure 4 (step ❸), TZER first selects an AST node (i.e., the 2 nested loops wrapped by the scope of `launch_thread`) and then initializes its threading parameters, e.g., virtual thread number (`virtual_thread` in TVM). In this way, `virtual_thread` is initialized by 2 which means this node will be executed by 2 virtual threads.

3.4 Test Oracle

Test oracles are important for detecting potential bugs with fuzzing. In this paper, we consider the following ways to resolve the test oracle problem for finding bugs in tensor compilers:

Result Inconsistency. Tzer holds the hypothesis that an IR, whether it is optimized or not, should keep the output result consistent. For each generated IR, Tzer will compile it twice, where it first compiles the IR with the lowest optimization and then compiles it with given optimization passes. In this way, Tzer compares the output results by feeding 2 model binaries the same input data. We identify it as an inconsistency bug if the absolute or relative error exceeds the expectation.

Performance Degradation. The second hypothesis by Tzer is that after a series of optimization passes, the performance should not be degraded. Therefore, Tzer would instrument the running time of optimized and non-optimized executions. If the optimized code runs even slower than the non-optimized one, we consider it as a potential performance bug. Notably, to avoid false-positives, we set clear performance margin in the differential testing setting. The non-optimized version is compiled with lowest optimization level (`opt_level=0`) while the optimized one is compiled with highest optimization level (`opt_level=4`). Note that higher optimization level allows better and more aggressive optimization than lower levels given the same pass sequences. For example, level-3 graph fusion (i.e., `FuseOps`) allows more operator fusion patterns than the low-level one.

Crash and Unexpected Exception. Like most Python applications, throwing an exception is the default behavior of errors. Hence, Python/C++ projects (e.g., most tensor compilers) need to convert C++ exceptions into Python ones. For example, in TVM's C++ codebase, any unexpected behavior (e.g., assertion failure) will result in C++ exceptions, where the top-level foreign function interface (FFI) handler will catch such C++ exceptions and pack the error message using the type `TVMError` for Python front-end. Therefore, though errors might occur, the symptom should be uncaught exceptions rather than crash. The compilation and execution phase of Tzer is done by forking a sub-process, Tzer observes such crash by checking the return code of sub-processes. Tzer also monitors exceptions thrown during compilation as potential bugs. To avoid false alarms, Tzer has made the best effort on constructing legal IRs and pass sequences.

4 EXPERIMENTAL SETUP

4.1 Research Questions

In this paper, we study the following research questions to thoroughly evaluate Tzer:

- **RQ1:** How is the effectiveness of Tzer compared with state-of-the-art fuzzing techniques on testing the TVM tensor compiler?
- **RQ2:** Are all components of Tzer contributing positive improvements to its final effectiveness?
- **RQ3:** How do different parameter settings and experimental setups affect Tzer's effectiveness?
- **RQ4:** How effective is Tzer in detecting previously unknown bugs?

The consideration of our experiment design largely follows suggestions made by Klees et al. [2018]. The main differences are caused by the fuzzing targets, i.e., Klees et al. [2018] mainly studied binary fuzzing while we are working on tensor compiler fuzzing. For example, the paper suggested a 24-hour timeout, while we evaluate Tzer with a default 4h timeout since existing techniques tend to saturate within 4 hour. Meanwhile, we do evaluate Tzer with a 24-hour budget as well in RQ3.

4.2 Implementation

Tzer has been mainly implemented in 8.7k lines of Python code and ~150 lines of C++ code for coverage extension with the following main components:

Mutators. We implemented all the 3 general-purpose mutators and 3 domain-specific mutators via directly operating on TIR in-memory objects (i.e., `tir.PrimFunc`) for fast mutation. More specifically,

the mutation procedure is implemented by extending the visitor pattern of TIR's recursive post-order traversal interface. In addition, the utility generator used by replacement and insertion is capable of constructing various sub-expressions based on 89 TIR operator APIs. When inserting/replacing sub-expressions into an existing TIR, we consider the syntactic/semantic correctness by maintaining IR constraints during the visiting process (e.g., preventing the use of variables that are undeclared or out of the scope). We further utilize casting nodes when generating intrinsic function calls. Although casting is not necessary theoretically due to our constraints-based approach, TVM provides more than 30 intrinsics whose detailed function signatures may vary and are not documented (e.g., `tir.cos` returns `float` whereas `tir.clz` returns `int`). To save manual efforts, we simply regard those intrinsics as opaque ones and cast them to satisfy the constraints.

Executor. Once Tzer generates a TIR file and pass sequence pair, they are sent to a sub-process for compilation and execution. The sub-process mechanism is to provide process-level isolation so that the fuzzing loop continues even though the TIR file and pass sequence make the sub-process crash.

Coverage Collector. We implemented `memcov`, our in-memory coverage instrumentation tool, by extending LLVM's Coverage Sanitizer (i.e., injecting a customized function when entering each of CFG edges in the target program). Once a program is compiled along with `memcov`, we maintain a bit vector whose size is exactly the number of CFG edges of the instrumented program (i.e., TVM). When entering one edge, its corresponding position on the bit vector is set to `True`. As we implemented Tzer's core components in Python, we also provide a Python interface to get the coverage state at that point by invoking C++ functions through `ctypes` [Foundation 2021] (a Python-C++ FFI tool).

Reporter. Once a test violates our test oracle, the reporter would record necessary contextual data to reproduce the failure and debugging.

Consistent with Algorithm 1, the Tzer implementation takes three inputs, i.e., S_0, T, and N_{max}. For the initial seed pool S_0, by default Tzer uses 629 TIR functions converted from all possible official models from TVM's model zoo (`tvm.relay.testing`); for the time budget T, by default Tzer sets it to 4 hours; for the IR-pass mutation control N_{max}, by default Tzer sets it to 5. We use such default setting for Tzer unless explicitly specified, e.g., we will present the detailed impacts of different parameter settings on Tzer in RQ2 (§ 5.2).

The main techniques behind Tzer are general to other tensor and even traditional compilers which model low-level IRs and optimization passes. To implement our approaches for a new compiler, one needs to implement language mutators following rules described in § 3.2 and § 3.3, as well as figuring out corresponding optimization passes. The syntactic and semantic correctness of mutated IRs and passes should also be maintained. After that, the main algorithm and skeleton of Tzer shall directly apply.

4.3 Compared Work

To faithfully evaluate the effectiveness of Tzer, we compare Tzer with both the state-of-the-art general-purpose fuzzers and domain-specific fuzzers that can be applied/adapted for TVM fuzzing. More specifically, we include the following representative techniques in our evaluation:

- TVMFuzz [Pankratz 2020]: This is the only existing fuzzer specifically targeting TVM to our knowledge. It follows a pure generation-based approach, which randomly generates TIR expressions by crafting valid expression ASTs of TIR. The generation approach is based on a user-defined probability table for different TIR nodes, while the validity is achieved by casting the input expressions to the parameter types of the operator.
- LibFuzzer [Serebryany 2016]: This is one of the state-of-the-art bit-level general-purpose binary fuzzers. It has been adopted as the first fuzzer supported by the famous Google OSS-Fuzz project [Serebryany 2017], which has found thousands of security vulnerabilities and stability

bugs; furthermore, it is also the officially used fuzzer for many popular projects including Chrome [Blog 2016] and glibc [Wiki 2016]. In this work, for a fair comparison with Tzer, we also run LibFuzzer with the TVM official model files (exported in JSON) as seeds for fuzzing TVM.

- LEMON [Wang et al. 2020]: LEMON is the state-of-the-art graph-level model generator for testing the operator-level DL libraries. At the graph level, different operators in a computation graph usually have various tensor shape constraints that are very complex to resolve. To resolve this difficulty, LEMON developed a series of mutators for shape-invariant operators and their compositions, by replacing operators with equivalent shape requirements or inserting/deleting element-wise operators. Since LEMON mutates the high-level computation graphs, its generated models can be directly applied to simulate TVM fuzzing at the high level. For a more fair comparison with LEMON, we also run a Tzer variant with LEMON's model seeds (this is because LEMON leverages Keras [Google 2015] model files which can be converted to TIR but cannot be done vice versa).

4.4 Metrics

We use the following metrics to evaluate the performance of Tzer and the compared techniques:
Code Coverage. Code coverage has been widely recognized as one of the most widely used metrics to evaluate software testing techniques [Gopinath et al. 2014]. The reason is that it is impossible for testing techniques to detect bugs in a code portion without actually executing it. Surprisingly, although existing work on testing deep learning libraries [Pham et al. 2019; Wang et al. 2020] claimed to cover more library code, they failed to present the detailed code coverage information. In this work, we instrument the entire TVM code base by extending LLVM's Coverage Sanitizer and collect the detailed code coverage information at the edge level for the studied techniques to thoroughly evaluate their test effectiveness. Note that since we are comparing techniques for fuzzing the TVM compilation process, to make the comparison fair, we omit the coverage brought by other irrelevant modules at the initialization phase (e.g., constructing TIR functions by converting input models).
Number of Valuable Tests. Following prior work on fuzzing [Park et al. 2020], for each compared technique, we also present the number of generated *valuable* tests, i.e., the tests that are not only valid (i.e., compilable) but also contribute new coverage during the fuzzing process. This metric is essential since the number of syntactically/semantically valid tests with new coverage can largely indicate the number of unique system behaviors/paths covered/tested. Also, this metric can largely complement code coverage, because techniques that mostly generate invalid inputs can still achieve high coverage for the error-handling code but that is clearly not what we want.
Number of Detected Bugs. Following almost all prior work on software testing and fuzzing [Li et al. 2018; Manès et al. 2019], we further present the number of previously unknown bugs detected by all the studied techniques since bug detection is the ultimate goal for such techniques. In this work, we distinguish different bugs based on how they are fundamentally fixed. For instance, we found that 21 TIR operator functions (such as `tir.op.clz(None)`) will crash when given NULL inputs on a specific TVM version, but we only count this as 1 bug since all the crashes can be fixed by changing only one C++ macro statement.

4.5 Experimental Procedure

For a fair comparison, we collect coverage of all compared techniques with the default 4-hour time budget using the same coverage collector that we implemented based on LLVM Coverage Sanitizer. Note that for TVMFuzz and other baselines requiring no coverage feedback, we first run them on non-instrumented TVM binary for 4 hours to prevent unnecessary overhead introduced by coverage tracing. Then, we collect the generated TIR files and passes (if any) from them, and compile them on instrumented TVM binary for offline coverage analysis. Notably, for LEMON, we collect the

Keras [Google 2015] models generated in 4 hours, and convert them to TIR functions. We then run the TIR functions on instrumented TVM to mimic the effectiveness of LEMON's graph-level construction for fuzzing TVM. Of course, for those studied techniques requiring coverage feedback, we directly record the coverage within one run on instrumented TVM.

We conducted experiments on: 1) *GPU test-bed*: a test-bed with Intel i9-9900X CPU (10 physical cores), GeForce RTX 2080 Ti GPU, and 128GB RAM, running 64-bit Ubuntu 18.04 as the operating system; and 2) *CPU test-bed*: a virtual cloud server (Alibaba Cloud ecs.c6e instance) with 4 CPU cores and 8GB RAM, running 64-bit Ubuntu 20.04. Since one of the baselines, LEMON, requires a GPU environment, we did RQ1 (comparison with existing work) on the *GPU test-bed* and all other RQs on the *CPU test-bed*. To ensure performance fairness, we made the system environment exclusive to the benchmarks so that the system average load is always around 1 during the process. For instrumentation, we compiled TVM v0.8-dev (`9b034d7`) with LLVM-12 and leveraged Coverage Sanitizer to trace edge coverage. TVM is compiled under optimization level `O2` and other configurations follow the default value. Since TVM contains as many as 17 targets, 4 executors, and many other irrelevant utilities (e.g., debuggers and profilers), in our evaluation, we focused on the LLVM-X86 target and the graph executor as they are widely adopted in tutorials and in practice.

5 RESULT ANALYSIS

5.1 RQ1: Comparison with Existing Work

Figure 5 presents the *coverage trends* for both Tzer and the compared existing work within the default 4-hour budget. To be specific, the x axis presents the time costs and the y axis shows the basic block coverage achieved. More powerful techniques are expected to achieve higher coverage at the same timestamp. As the figure shows, Tzer is able to beat other compared techniques at the very beginning and eventually achieves 75% higher coverage than the 2nd-best baseline (i.e., TVMFuzz). Notably, Tzer is able to keep visible coverage increase even at the late stage of the 4-hour budget while other techniques tend to converge very quickly. Another interesting observation is that Tzer with the same seeds as LEMON even achieves slightly higher coverage than the default Tzer, demonstrating the robustness of Tzer.

Table 2 further presents the number of *valuable tests* (i.e., the tests that are both compilable and able to trigger new coverage) generated by all the compared techniques within 4 hours. Regarding the comparison of graph-level and low-level IR mutations, Tzer is able to generate 7.7x more valuable tests than the state-of-the-art graph-level mutator LEMON. Specifically, LEMON only generates 63 valuable tests when the models are lowered to TIR functions (one model can be lowered to multiple TIR functions); if we had considered valuable tests at its original model level, the number of valuable tests is merely 20 out of all the 2.6k models generated by LEMON (i.e., 0.7%). We can also observe that LibFuzzer can hardly generate valid tests since it is a bit-level fuzzer, not aware of the grammar and semantics behind. Lastly, among the low-level IR fuzzers, Tzer is still able to outperform TVMFuzz by 50% in terms of valuable tests. The main reason is that TVMFuzz follows a pure generation-based approach (which lacks coverage guidance and makes it challenging to simulate realistic IRs) and does not consider the mutual effect of IR and pass combinations.

5.2 RQ2: Ablation Study of Tzer

In this RQ, we further study the effectiveness of Tzer's individual components:

(1) **RQ2.1**: Is coverage feedback helpful for tensor compiler fuzzing?
(2) **RQ2.2**: Can domain-specific mutations further improve tensor compiler fuzzing?
(3) **RQ2.3**: Are pass mutations necessary for tensor compiler fuzzing?

[2]If not specified, Tzer seeds (§ 4.5) are used by default. Initial seeds are not taken into account for fair comparison.

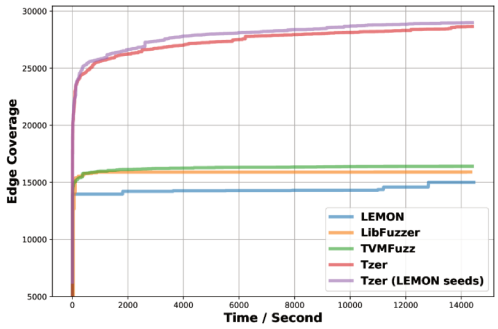

Fig. 5. Comparison with Existing Work

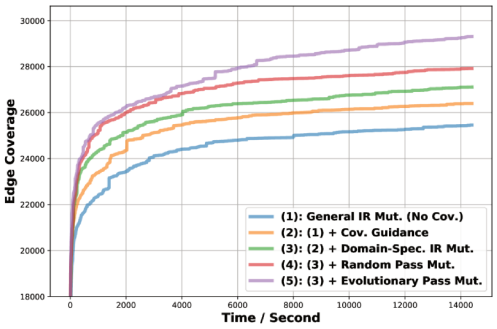

Fig. 6. Ablation Study of Tzer's Components

Table 2. Number of Generated Valuable Tests[2]

Tools	# Valuable Tests
Tzer	**497**
TVMFuzz	331
LibFuzzer	38
Tzer (LEMON seeds)	**485**
LEMON (LEMON seeds)	63

(4) **RQ2.4**: Can our evolutionary joint IR-pass mutation (described in § 3.1) outperform a baseline joint IR-pass mutation that mutates both IR and pass sequences simultaneously?

To answer the above questions, we first build a simplistic variant of Tzer that only applies general-purpose mutation (i.e., without coverage feedback, domain-specific mutation, or joint IR-pass mutation). Then, we incrementally add more components to the simplistic variant in the order of coverage feedback, domain-specific mutation, random joint IR-pass mutation, and evolutionary joint IR-pass mutation. Curves (1) to (5) in Figure 6 represent the coverage trends after adding each component progressively. From curves (1) and (2), we can see that coverage feedback has positive effects on tensor compiler fuzzing. Curves (2) and (3) confirm the effectiveness of domain-specific IR mutation in addition to general-purpose IR mutation. RQ2.3 can be answered by comparing curve (3) against curves (4) or (5), as extended pass sequence mutation could help trigger more interesting behaviors. Lastly, comparing curves (4) and (5), it can be shown that our evolutionary joint IR-Pass mutation is superior to the random joint IR-pass mutation, which performs coverage-guided fuzzing on IR files and supplies a randomly mutated pass sequence to each generated IR file. Hence, we can draw a conclusion that all the main components of Tzer contribute to tensor compiler fuzzing.

5.3 RQ3: Parameter Sensitivity

Sensitivity to Seeds (S_0) The first sub-figure in Figure 7 shows how Tzer performs with and without the default initial seed pool. Surprisingly, the non-seed version has comparable (and even slightly better at some time stamps) effectiveness to the default Tzer with 629 TIR seeds in terms of the coverage trend. This is because, though each iteration Tzer with seeds could generate higher-quality tests (the yellow curve is higher than the blue one in the 2nd sub-figure of Figure 7), the non-seed version runs 24% faster than that with seeds on average (as shown in the 3rd sub-figure). The rationale behind is that if initial seeds are not given, Tzer has to start IR mutation from an empty TIR function (i.e., `PrimFunc([]) {0}`) so that mutated variant IR files are similarly simple. Hence, the overall compilation time of simple IRs will be smaller than the complex ones derived from real models.

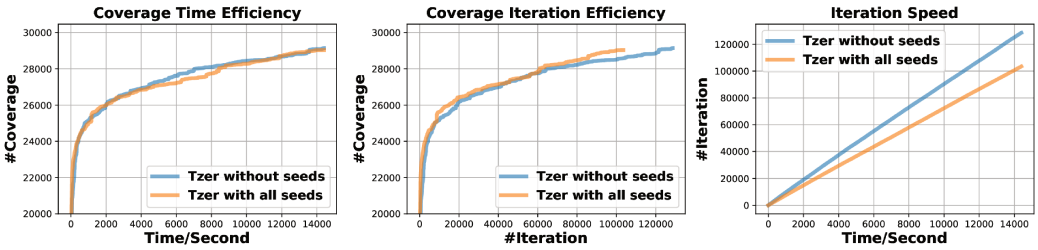

Fig. 7. Impact of Seeds on Tzer

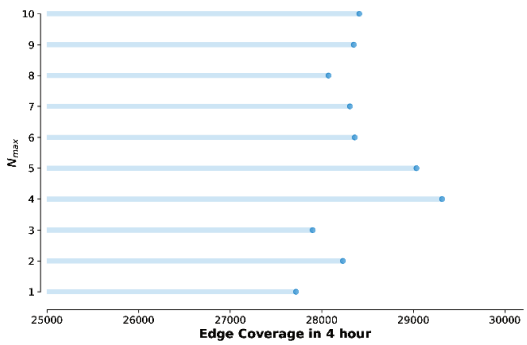

Fig. 8. Impact of Parameter N_{max} on Peak Coverage

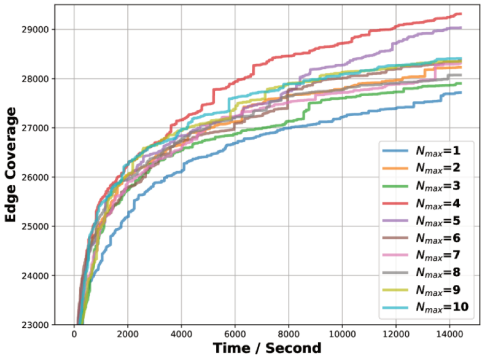

Fig. 9. Impact of Parameter N_{max} across Time

Sensitivity to Pass Mutation Frequency (N_{max}) Compared with traditional fuzzing loop, Tzer has an extra parameter N_{max} to control the IR-Pass mutation interleaving (see § 3.1). To study the impact of N_{max}, we conducted the experiment using different values from 1 to 10 for N_{max}. Figure 8 presents the final 4-hour coverage of different settings, while Figure 9 presents the corresponding detailed coverage trends. From the figures, we can see that $N_{max} = 4$ demonstrates the best effectiveness. In addition, $N_{max} = 1$ performs the worse in terms of the peak coverage and overall trend. This is because the coverage is mainly contributed by testing different IRs and the coverage growth will slow down if we frequently "freeze" the newly found IRs and mutate the pass sequences instead. We can also observe that the coverage does not keep growing if we keep increasing N_{max} (i.e., decreasing the probability of pass mutation). The rationale behind is that though pass mutation contributes less than IR mutation in the early stage, it is still important to mutate the pass sequence for an "old" IR that is not very likely to derive new interesting IRs anymore with its current pass sequence. In conclusion, while Tzer with different pass-mutation frequency values can all outperform existing state-of-the-art techniques, highly frequent pass mutations may not be cost-effective, while highly sparse pass mutations may miss the important chances to mutate the pass sequences of some IRs to trigger new coverage. Therefore, it is important to select a proper pass mutation frequency value (not too high or too low) to help Tzer achieve the best performance.

Sensitivity to Fuzzing Time (T) Figure 10 shows the overall coverage trend achieved by the default Tzer across 24 hours. While the existing techniques already saturate within 4 hours (shown in RQ1), Tzer is able to successively keep coverage growth for the entire 24-hour period. Specifically, the first 4-hour window contributes the most coverage, i.e., 91.6%, while later 5 4-hour windows are still able to contribute 2.1%, 2.8%, 1.9%, 1.1%, and 0.5% coverage respectively, demonstrating the effectiveness of Tzer.

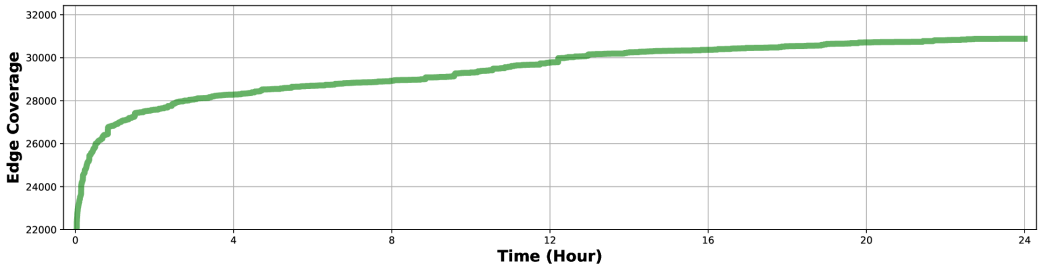

Fig. 10. 24-hour Coverage Trend of Tzer

In terms of the total code coverage of TVM, at the source code level, Tzer can at best achieve 36.9% line coverage and 28% branch coverage with 4 CPU hours by only tracing the source files used in normal compilation. At the LLVM bitcode level, there are 482k CFG edges in total for our target, and Tzer achieves about 6% coverage within 4 CPU hours. This is because LLVM coverage sanitizer takes code bloating into account (C++ headers, templates and inlined functions are repeatedly considered) and thus can present underestimated coverage rates. Also, please note that modest overall coverage rates are quite common for fuzz testing of complicated software systems. For example, existing state-of-the-art Linux kernel fuzzers implement coverage collection with LLVM as well. Although they do not suffer from template code bloating in C++ as Linux is mostly implemented in C, the fuzzers can only achieve 0.8∼10.5% coverage after 50-hour fuzzing by fully utilizing a 32-core high-end CPU [Kim et al. 2020].

5.4 RQ4: Bug Detection Effectiveness

To date, Tzer has found 49 previously unknown unique bugs. Table 3 shows the detailed information about the 37 bugs that have been confirmed by TVM's developers, where 25 of them have already been fixed and merged to the main branch of TVM.

Tzer generates tests through pass mutation, IR mutation, and their combination. It is important to understand the necessity and effectiveness of each part. Table 4 further presents the overall statistics for the bugs and bug types (categorized based on bug root causes) found by different studied techniques. In terms of confirmed bugs, we can find that in addition to only mutating IRs (i.e., Column "Tzer-IR"), modelling IR/Pass jointly (i.e., Column "Tzer-Full") could help detect 2.17x more bugs and 1.6x more bug types. Existing fuzzers for compilers, not limited to tensor compilers, only consider the compiler under test as a black box ingesting input source language texts and ignore the mutual effect of IRs and pass sequences internally applied together. Tzer demonstrates for the first time that it could be beneficial to perform evolutionary joint IR-Pass mutation for better and deeper bug detection.

From Table 4, we can also observe that bugs detected by Tzer can hardly be detected by other compared techniques, e.g., Tzer detects 6.16x more confirmed bugs compared with the 2nd-best technique, TVMFuzz. This is mainly because Tzer has a more complete modelling for both IR and pass sequences, as well as having a better fuzzing efficiency to quickly harness the large well-modelled search space (with coverage guidance). In addition, according to Figure 5, Tzer is able to consistently find uncovered CFG edge while other techniques converge at a very early stage, which explains why existing techniques fail to help discover more potential bugs.

5.5 Bug Root Causes and Case Study

To demonstrate the versatility of Tzer, we study root causes of confirmed bugs detected by Tzer as shown in Table 3, and discuss representative bugs for each category:

Table 3. Summary of Confirmed Bugs Detected by Tzer in TVM (v0.8-dev). Abbreviations like "31~36" represent multiple different bugs instead of one bug.

API-I: API Inconsistency (§ 5.5.5). **API-M**: API Misuse (§ 5.5.4). **AE**: Arithmetic Error (§ 5.5.9).
DL: Driver Lifetime (§ 5.5.7). **FFI**: Foreign Function interface (§ 5.5.2). **IMA**: Invalid Memory Access (§ 5.5.1).
OOM: Out Of Memory (§ 5.5.8). **PMI**: Pass-Module Immutability (§ 5.5.3). **TE**: Type Error (§ 5.5.6).

ID	Root Cause	Triggering Components			Symptom	Status
		IR	Pass	Runtime		
1	TE	✓			Crash	Fixed
2	IMA	✓			Crash	Fixed
3	IMA	✓			Crash	Fixed
4	TE	✓			Exception	Fixed
5~7	API-M		✓		Performance	Fixed
8	API-I		✓		Exception	Fixed
9	PMI		✓		Inconsistency	Confirmed
10	FFI	✓			Crash	Fixed
11	API-I		✓	✓	Crash	Fixed
12	API-I	✓	✓		Exception	Fixed
13	IMA	✓			Crash	Fixed
14	TE	✓			Crash	Fixed
15	IMA	✓	✓		Crash	Fixed
16	IMA	✓	✓		Crash	Fixed
17	FFI	✓			Crash	Fixed
18	FFI	✓			Crash	Fixed
19	IMA	✓	✓		Crash	Fixed
20~23	AE	✓	✓		Crash	Fixed
24~26	IMA	✓			Crash	Confirmed
27	DL			✓	Crash	Fixed
28	OOM			✓	Crash	Fixed
29	N/A			✓	Exception	Confirmed
30	IMA			✓	Crash	Confirmed
31~36	AE	✓	✓		Crash	Confirmed
37	AE	✓	✓		Crash	Fixed

Table 4. Detectable Confirmed Bugs by Different Methods and Tzer Components

Methods	LEMON	TVMFuzz	LibFuzzer	Tzer-IR	Tzer-Full
#Valid bugs	3	6	3	17	37
#Bug Type	3	5	3	6	10

5.5.1 Invalid Memory Access. Since most tensor compilers leverage memory-unsafe languages (i.e., C/C++) to implement the core components, it is not surprising that they will suffer from various memory problems, like out-of-bound access or NULL pointer dereference. Invalid memory access is one of the most frequently detected bug types by Tzer. For example, Tzer discovered an out-of-bound access bug triggered by a specific combination of IR and pass. When applying pass `InjectVirtualThread` on an IR module, the IR would be converted to the SSA format first, traversing all expressions to create a variable-to-array mapping for recording variable status. Theoretically, an array might not exist by variable name or it is temporarily empty. However, Tzer found during visiting `Save` or `Load` expressions, the TVM compiler only checks the existence of corresponding array and then directly accesses the last element (i.e., `std::vector::back` in C++) without boundary checking, resulting in a crash. We illustrate a simplified fix to it in Listing 1.

In addition to an out-of-bound access to containers, a crash occurs if a NULL pointer is dereferenced. According to the TVM design, objects could be nullable (an optional type containing a NULL state) or

```
PrimExpr VisitExpr_(const LoadNode* op) final {
......
-   if (scope_.count(v)) {
+   if (scope_.count(v) && !scope_[v].empty()) {
    return Load(op->dtype, scope_[v].back(), op->index, op->predicate);
```

Listing 1. Sample fix for Missing Boundary Checking

non-nullable. In TVM, an object accesses its data members or member functions by the `->` operator in C++, which assumes any objects using operator `->` are not NULL objects. However, even for nullable objects, Tzer found over 44 functions (categorized as 3 unique bugs) do not check if a receiving nullable object is NULL or not, resulting in immediate crashes in case of NULL objects.

5.5.2 Python-C++ FFI Handling. Same as most other deep learning software, TVM and most other tensor compilers provide a Python interface, i.e., Foreign Function Interface (FFI), to bind Python functions and objects to C++ functions and objects through the `ctypes` standard library [Foundation 2021] and `cython` [Behnel et al. 2010]. The motivation is that most deep learning practitioners are familiar with Python instead of C++. However, python requires objects to support numerous built-in functions. For example, Tzer found the `StringImm` object in TVM failed to provide a `__hash__` implementation and threw an unexpected exception when put into a map container.

5.5.3 Pass-Module Immutability. TVM's passes mark the input IR module as `const` object (i.e., `const IRModule`), meaning that member functions that mutate data members cannot be called by such objects. However, Tzer found a pass, i.e., `ToBasicBlockNormalForm`, violating this contract by permitting the input `const` IR object to call non-`const` methods using pointers (C++ codebase), resulting in inconsistency issues in Python front-end. We fixed this bug by forcing a copy at the beginning of the transformation. A simplified bug fix is shown in Listing 2.

```
- IRModule ToBasicBlockNormalForm(const IRModule& mod) {
+ IRModule ToBasicBlockNormalForm(const IRModule& mod_) {
+   auto mod = IRModule(mod_->functions,
↪  mod_->type_definitions, mod_->Imports(), mod_->source_map); // Make 'mod' a copied object.
```

Listing 2. Sample Fix for the Pass-Module Immutability Bug

5.5.4 API Misuse. Tzer also surprisingly detected that sometimes O4 optimization performs even worse than O2 (default optimization). This is actually because we followed TVM's official tutorial when building Tzer while their tutorial misused the API which failed to invoke the desired optimization. In TVM's Python API, optimization level can be be identified within a scope called `PassContext` (Line 1 in Listing 3). In Listing 3, old tutorial code calls `.evaluate()` outside the `PassContext` scope. The `evaluate()` function, nevertheless, is where the optimizations are applied. Therefore, when calling `evaluate` out of the O4 scope, the default optimization (O2) will be applied so that when comparing with another O2-optimized binary (they are all equally optimized), it is possible to see one is slower than the other due to uncertainty.

5.5.5 API Inconsistency. Inconsistency in API happens when a program does not act as what the API is specified. For example, when running programs on heterogeneous devices (e.g., run a program that requires both GPU and CPU), TVM splits the functions into either the host side or the device side. There is a parameter controlling the calling convention (i.e., `calling_conv`) for the heterogeneous

```
with tvm.transform.PassContext(opt_level=4):
-   executor = relay.build_module.create_executor("graph", mod, dev, target)
+   executor = relay.build_module.create_executor("graph", mod, dev, target, params).evaluate()
.......
- tvm_out = executor.evaluate()(tvm.nd.array(data.astype(dtype)), **params)
+ tvm_out = executor(tvm.nd.array(data.astype(dtype)))
```

Listing 3. Sample Fix to API Misuse

compilation, which is set to `kDefault` by default. `kDefault` generally means that both host and device targets are CPUs (e.g., LLVM as the target). However, a pass called `DecorateDeviceScope` violates the calling convention by implicitly change `kDefault` into `kDeviceKernelLaunch` which is built for non-CPU device targets (i.e., the `DecorateDeviceScope` is not desired to change the calling convention). Such an inconsistency leads to a crash at runtime.

5.5.6 Type Error. Tzer found an issue regarding TVM's constant folding in integer conversion. For example, the expression `assert tir.const(1) == tir.const(True)` would throw unexpected exceptions, whereas we expected it to be a `True` after evaluation. The root cause is that conversion for signed/unsigned integers (int64 and boolean) is not well handled. Theoretically, since the range of boolean type is the subset of int64's, we can convert the boolean value to an int64 value. We fundamentally fixed the issue by refining TVM's type conversion for signed and unsigned integers.

5.5.7 Driver Lifetime Error. Tzer found that when enabling CuDNN [Chetlur et al. 2014] as the target backend, TVM crashes after being stuck for a while when the program exits. This is because TVM made the CuDNN device handler of a whole-process lifetime by marking it with `thread_local` (a specifier in C++). Thus, according to the RAII rule [Stroustrup 2017] of C++, the deconstructor to release the handler will be called during program exit. However, the CuDNN library context might have already been exited when such release handlers are being called, causing segmentation fault after a long suspension.

We further proposed 2 fixes to this problem: (1) we register the handler release function at exit time using `atexit`, and make sure that the destroyer of library context will be called after it; (2) we simply remove the handler release code and let it leak since we do not need to do recycling when a program is going to exit. The community finally accepted proposal (2) since proposal (1) is more advanced and complex, increasing the maintenance cost.

5.5.8 Out-of-Memory. Tzer found an interesting out-of-memory (OOM) bug when using the virtual machine (VM) as TVM's executor. The cause is that the previous VM memory allocator never releases occupied memory in the memory pool and leverages no memory defragmentation strategies. It only re-uses memory blocks in the pool if the incoming request size is smaller than the existing one. When the memory requests follow a monotonic pattern, it fails eventually since it cannot release previous memory blocks in the pool.

For example, as shown in Table 5, on a GPU of 8 GB memory, if for each time, we release i GB memory and allocate i+1 GB memory (i starts from 0), it will fail in the 4th step. The reason is that each time, after releasing i GB memory, the released memory chunk is returned to the free list; when requesting i+1 GB next time, all chunks in the pool cannot be used since they are smaller than i+1 GB. Hence, in the 4th step, even though the GPU has 8 GB physical memory, it cannot allocate a 4 GB memory chunk.

We fixed this issue by simply releasing all cache blocks and re-attempting allocation if any OOM exceptions are caught.

Table 5. Example of How TVM's VM Allocator Failed in Monotonic Allocation.

#Iter.	Next Action	Avai. System Mem.	Occupied Mem.	Free list (pool)
1	Alloc. 1 GB	8 GB	0	empty
2	Free 1 & alloc. 2 GB	7 GB	1 GB	empty
3	Free 2 & alloc. 3 GB	5 GB	2 GB	[1] GB
4	Free 3 & alloc. 4 GB (Fail)	2 GB	3 GB	[1, 2] GB

5.5.9 Arithmetic Error. Tzer also found some functions in TVM fail to check the legality of arithmetic operations, such as division by zero. This bug lies in an optimization that simplifies the calculation of TIR. Specifically, when TVM tries to simplify a division expression whose two operands are of type Ramp and Broadcast, it will directly modulo two numbers without checking the divisor. This causes the program to crash when the divisor is 0.

6 CONCLUSION

The evolution of tensor compilers requires automated testing to achieve high maintainability and reliability. We demonstrate that existing fuzzing techniques are not tailored or effective enough to fulfill this mission. To this end, we present Tzer, a practical coverage-guided tensor compiler fuzzer with joint IR-Pass mutation. Unlike traditional compiler fuzzers, Tzer performs joint IR and pass mutation to explore various program states and introduces coverage guidance to navigate the mutation process. Specifically, in addition to general-purpose mutators, Tzer also leverages tailored domain-specific mutators to target the hotspot logics behind tensor compilers. The evaluation shows that Tzer substantially outperforms the state-of-the-art fuzzers including a general-purpose fuzzer (i.e., LibFuzzer), a graph-level DL model fuzzer (i.e., LEMON), and the only domain-specific fuzzer for TVM (i.e., TVMFuzz). As one of the practical contributions of Tzer, to date, we have helped the TVM community find 49 new unique bugs, with 37 confirmed and 25 of them already fixed in the current TVM version. Our effort has been highly recognized by the TVM community, and the leading author of Tzer has been nominated as a community reviewer for TVM.

7 DATA AVAILABILITY STATEMENT

Tzer has been open-sourced at GitHub (https://github.com/ise-uiuc/tzer) with the source code, experimental data used in this paper and documents. The artifact [Liu et al. 2022] is also available on Zenodo and the detailed instructions for reproducing the results can be found at Tzer's documentation (tzer.rtfd.io/en/latest/markdown/artifact.html).

ACKNOWLEDGMENTS

This work is partially supported by National Science Foundation under Grant Nos. CCF-2131943 and CCF-2141474.

REFERENCES

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 265–283. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. 2010. Cython: The best of both worlds. *Computing in Science & Engineering* 13, 2 (2010), 31–39.

Petter Bjørstad, Fredrik Manne, Tor Sørevik, and Marian Vajteršic. 1992. Efficient matrix multiplication on SIMD computers. *SIAM J. Matrix Anal. Appl.* 13, 1 (1992), 386–401.

Google Security Blog. 2016. Guided in-process fuzzing of Chrome components. https://security.googleblog.com/2016/08/guided-in-process-fuzzing-of-chrome.html.

Marcel Böhme, Valentin JM Manès, and Sang Kil Cha. 2020. Boosting fuzzer efficiency: An information theoretic perspective. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 678–689.

Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2017. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering* 45, 5 (2017), 489–506.

Zhe Cao, Gines Hidalgo, Tomas Simon, Shih-En Wei, and Yaser Sheikh. 2019. OpenPose: realtime multi-person 2D pose estimation using Part Affinity Fields. *IEEE transactions on pattern analysis and machine intelligence* 43, 1 (2019), 172–186.

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.

Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759* (2014).

Koen Claessen, Jonas Duregård, and Michał H Pałka. 2015. Generating constrained random data with uniform distribution. *Journal of functional programming* 25 (2015).

Apache TVM Community. 2020. tvm.relay.testing — tvm 0.8.dev0 documentation. https://tvm.apache.org/docs/api/python/relay/testing.html.

Maxime Dénès, Catalin Hritcu, Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C Pierce. 2014. QuickChick: Property-based testing for Coq. In *The Coq Workshop*, Vol. 125. 126.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, 4171–4186. https://doi.org/10.18653/v1/n19-1423

Andre Esteva, Alexandre Robicquet, Bharath Ramsundar, Volodymyr Kuleshov, Mark DePristo, Katherine Chou, Claire Cui, Greg Corrado, Sebastian Thrun, and Jeff Dean. 2019. A guide to deep learning in healthcare. *Nature medicine* 25, 1 (2019), 24–29.

Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association.

Python Software Foundation. 2021. https://docs.python.org/3/library/ctypes.html.

Joshua Garcia, Yang Feng, Junjie Shen, Sumaya Almanee, Yuan Xia, and Qi Alfred Chen. 2020. A comprehensive study of autonomous vehicle bugs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 385–396.

Google. 2015. Keras. https://keras.io.

Google. 2016. XLA: Optimizing Compiler for Machine Learning. https://www.tensorflow.org/xla.

Rahul Gopinath, Carlos Jensen, and Alex Groce. 2014. Code coverage for suite evaluation by developers. In *Proceedings of the 36th International Conference on Software Engineering*. 72–82.

Sorin Grigorescu, Bogdan Trasnea, Tiberiu Cocias, and Gigel Macesanu. 2020. A survey of deep learning techniques for autonomous driving. *Journal of Field Robotics* 37, 3 (2020), 362–386.

Yixiao Guo, Jiawei Liu, Guo Li, Luo Mai, and Hao Dong. 2021. Fast and Flexible Human Pose Estimation with HyperPose. *arXiv preprint arXiv:2108.11826* (2021).

Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *21st USENIX Security Symposium (USENIX Security 12)*. USENIX Association, Bellevue, WA, 445–458. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler

Intel. 2017. PlaidML is a framework for making deep learning work everywhere. https://github.com/plaidml/plaidml.

Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: Optimizing Deep Learning Computation with Automatic Generation of Graph Substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 47–62. https://doi.org/10.1145/3341301.3359630

Tian Jin, Gheorghe-Teodor Bercea, Tung D Le, Tong Chen, Gong Su, Haruki Imai, Yasushi Negishi, Anh Leu, Kevin O'Brien, Kiyokuni Kawachiya, et al. 2020. Compiling ONNX Neural Network Models Using MLIR. *arXiv preprint arXiv:2008.08272* (2020).

Kyungtae Kim, Dae Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. 2020. HFL: Hybrid Fuzzing on the Linux Kernel. https://doi.org/10.14722/ndss.2020.24018

George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 2123–2138. https://doi.org/10.1145/3243734.3243804

Sven Kreiss, Lorenzo Bertoni, and Alexandre Alahi. 2019. PifPaf: Composite Fields for Human Pose Estimation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 11969–11978. https://doi.org/10.1109/CVPR.2019.01225

Leonidas Lampropoulos, Michael Hicks, and Benjamin C Pierce. 2019. Coverage guided, property based testing. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–29.

Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C Pierce. 2017. Generating good generators for inductive relations. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–30.

Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2020. MLIR: A compiler infrastructure for the end of Moore's law. *arXiv preprint arXiv:2002.11054* (2020).

Chris Arthur Lattner. 2002. *LLVM: An infrastructure for multi-stage optimization.* Ph.D. Dissertation. University of Illinois at Urbana-Champaign.

Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. *ACM Sigplan Notices* 49, 6 (2014), 216–226.

Caroline Lemieux and Koushik Sen. 2018. *FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage.* Association for Computing Machinery, New York, NY, USA, 475–485. https://doi.org/10.1145/3238147.3238176

Jun Li, Bodong Zhao, and Chao Zhang. 2018. Fuzzing: a survey. *Cybersecurity* 1, 1 (2018), 1–13.

Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. 2020. The deep learning compiler: A comprehensive survey. *IEEE Transactions on Parallel and Distributed Systems* 32, 3 (2020), 708–727.

Jiawei Liu, Yuxiang Wei, Sen Yang, Yinlin Deng, and Lingming Zhang. 2022. *Coverage-Guided Tensor Compiler Fuzzing with Joint IR-Pass Mutation.* https://doi.org/10.5281/zenodo.6371291

Valentin Jean Marie Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. 2019. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering* 47 (2019), 2312–2331.

Riccardo Miotto, Fei Wang, Shuang Wang, Xiaoqian Jiang, and Joel T Dudley. 2018. Deep learning for healthcare: review, opportunities and challenges. *Briefings in bioinformatics* 19, 6 (2018), 1236–1246.

Augustus Odena, Catherine Olsson, David Andersen, and Ian Goodfellow. 2019. TensorFuzz: Debugging Neural Networks with Coverage-Guided Fuzzing. In *Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, 4901–4911. https://proceedings.mlr.press/v97/odena19a.html

David Pankratz. 2020. TVMFuzz: Fuzzing Tensor-level Intermediate Representation in TVM. https://github.com/dpankratz/TVMFuzz.

Neungsoo Park, Bo Hong, and Viktor K Prasanna. 2003. Tiling, block data layout, and memory hierarchy performance. *IEEE Transactions on Parallel and Distributed Systems* 14, 7 (2003), 640–654.

Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. 2020. Fuzzing JavaScript Engines with Aspect-preserving Mutation. In *2020 IEEE Symposium on Security and Privacy (SP)*. 1629–1642. https://doi.org/10.1109/SP40000.2020.00067

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019), 8026–8037.

Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2019. DeepXplore: Automated Whitebox Testing of Deep Learning Systems. *Commun. ACM* 62, 11 (oct 2019), 137–145. https://doi.org/10.1145/3361566

Hung Viet Pham, Thibaud Lutellier, Weizhen Qi, and Lin Tan. 2019. CRADLE: Cross-backend validation to Detect and Localize bugs in Deep learning libraries *(ICSE '19)*. IEEE Press, 1027–1038. https://doi.org/10.1109/ICSE.2019.00107

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices* 48, 6 (2013), 519–530.

Qing Rao and Jelena Frtunikj. 2018. Deep Learning for Self-Driving Cars: Chances and Challenges. In *2018 IEEE/ACM 1st International Workshop on Software Engineering for AI in Autonomous Systems (SEFAIAS)*. 35–38.

Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, et al. 2018. Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907* (2018).

Kosta Serebryany. 2016. Continuous fuzzing with libfuzzer and addresssanitizer. In *2016 IEEE Cybersecurity Development (SecDev)*. IEEE, 157–157.

Kostya Serebryany. 2017. OSS-Fuzz-Google's continuous fuzzing service for open source software. (2017).

Tyler M Smith, Robert Van De Geijn, Mikhail Smelyanskiy, Jeff R Hammond, and Field G Van Zee. 2014. Anatomy of high-performance many-threaded matrix multiplication. In *2014 IEEE 28th International Parallel and Distributed Processing*

Symposium. IEEE, 1049–1059.

Bjarne Stroustrup. 2017. Why doesn't C++ provide a "finally" construct? https://www.stroustrup.com/bs_faq2.html#finally.

Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th international conference on software engineering*. 303–314.

Philippe Tillet, H. T. Kung, and David Cox. 2019. *Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations*. Association for Computing Machinery, New York, NY, USA, 10–19. https://doi.org/10.1145/3315508.3329973

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

Zan Wang, Ming Yan, Junjie Chen, Shuang Liu, and Dongdi Zhang. 2020. 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*. 788–799.

Anjiang Wei, Yinlin Deng, Chenyuan Yang, and Lingming Zhang. 2022. Free Lunch for Testing: Fuzzing Deep-Learning Libraries from Open Source. *arXiv preprint arXiv:2201.06589* (2022).

Glibc Wiki. 2016. Fuzzing libc. https://sourceware.org/glibc/wiki/FuzzingLibc.

Kanit Wongsuphasawat, Daniel Smilkov, James Wexler, Jimbo Wilson, Dandelion Mane, Doug Fritz, Dilip Krishnan, Fernanda B Viégas, and Martin Wattenberg. 2017. Visualizing dataflow graphs of deep learning models in tensorflow. *IEEE transactions on visualization and computer graphics* 24, 1 (2017), 1–12.

Mingyuan Wu, Ling Jiang, Jiahong Xiang, Yanwei Huang, Heming Cui, Lingming Zhang, and Yuqun Zhang. 2022. One Fuzzing Strategy to Rule Them All. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*.

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 283–294. https://doi.org/10.1145/1993498.1993532

Tom Young, Devamanyu Hazarika, Soujanya Poria, and Erik Cambria. 2018. Recent trends in deep learning based natural language processing. *ieee Computational intelligenCe magazine* 13, 3 (2018), 55–75.

Michal Zalewski. 2018. American Fuzzing Lop (AFL). https://lcamtuf.coredump.cx/afl/.

Mengshi Zhang, Yuqun Zhang, Lingming Zhang, Cong Liu, and Sarfraz Khurshid. 2018. DeepRoad: GAN-based metamorphic testing and input validation framework for autonomous driving systems. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 132–142.

Qirun Zhang, Chengnian Sun, and Zhendong Su. 2017. Skeletal Program Enumeration for Rigorous Compiler Testing. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 347–361. https://doi.org/10.1145/3062341.3062379

Jie Zhao, Bojie Li, Wang Nie, Zhen Geng, Renwei Zhang, Xiong Gao, Bin Cheng, Chen Wu, Yun Cheng, Zheng Li, Peng Di, Kun Zhang, and Xuefeng Jin. 2021. AKG: Automatic Kernel Generation for Neural Processing Units Using Polyhedral Transformations *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 1233–1248. https://doi.org/10.1145/3453483.3454106

Yingquan Zhao, Zan Wang, Junjie Chen, Mengdi Liu, Mingyuan Wu, Yuqun Zhang, and Lingming Zhang. 2022. History-Driven Test Program Synthesis for JVM Testing. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*.

Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. 2020. Squirrel: Testing database management systems with language validity and coverage feedback. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 955–970.

Husheng Zhou, Wei Li, Zelun Kong, Junfeng Guo, Yuqun Zhang, Bei Yu, Lingming Zhang, and Cong Liu. 2020. DeepBillboard: Systematic Physical-World Testing of Autonomous Driving Systems. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 347–358. https://doi.org/10.1145/3377811.3380422