DeepDiffer: Find Deep Learning Compiler Bugs via Priority-guided Differential Fuzzing

Kuiliang Lin[1], Xiangpu Song[1], Yingpei Zeng[2,*], and Shanqing Guo[1,*]
[1]Shandong University, Qingdao, Shandong, China
[2]Hangzhou Dianzi University, Hangzhou, Zhejiang, China
linkuiliang@mail.sdu.edu.cn, songxiangpu@mail.sdu.edu.cn, yzeng@hdu.edu.cn, guoshanqing@sdu.edu.cn
*corresponding author

Abstract—Recently, Deep learning (DL) compilers have been widely developed to optimize the deployment of DL models. These DL compilers transform DL models into high-level intermediate representation (IR) and then into low-level IR, ultimately generating optimized codes for different hardware targets. However, DL compilers are not immune to generating incorrect code, leading to potentially severe consequences. Testing techniques for low-level IR are limited, and efficient approaches for detecting some categories of non-crashing bugs are lacking. In this paper, we address the limitations of existing low-level IR DL compiler testing techniques and introduce DeepDiffer, a priority-guided differential testing framework designed to detect bugs resulting from low-level optimizations in the DL compiler, specifically TVM. We propose a novel DL compiler coverage metric and establish an optimization goal to maximize the detection of valuable differences between DL compilers. Our experiments demonstrate that DeepDiffer outperforms existing low-level IR fuzzers, detecting a wider range of bug types. In fact, DeepDiffer has successfully identified 13 bugs in TVM, which can be categorized into 9 distinct root causes, and 9 bugs are first found. We have submitted these bugs to the TVM community, where they have been confirmed.

Keywords–Fuzzing; Differential Testing; Compiler Testing; Machine Learning Systems

1. Introduction

Deep learning (DL) compilers, including TVM [1], Glow [2], and nGraph [3], play a crucial role in optimizing deep learning models to meet the specific requirements of deployment on various devices. These compilers take a deep learning model as input and apply multiple optimizations to generate hardware-optimized code as output [4]. However, like traditional compilers, deep learning compilers are susceptible to bugs, which can lead to a range of unexpected behaviors such as crashes, poor performance, and incorrect code generation [4]. These undesired behaviors can significantly impact the accuracy and reliability of deep learning applications. Moreover, the complexity of compiler implementations poses challenges for diagnosing errors in deep learning compilers.

Designing automated testing techniques for deep learning compilers is essential. Fuzzing has emerged as a widely used technique for identifying software bugs and testing compilers [5], [6], [7]. However, generic binary fuzzers [8], [9], [10] face challenges in generating valid inputs and parsing the components of DL models due to the distinct characteristics between traditional compilers and DL compilers [11]. DL compilers encompass multiple optimization stages, including high-level and low-level optimizations. Recent research [12] has made efforts to generate models guided by constraints to detect high-level bugs. Nonetheless, extracting these constraints is challenging, and its search space is limited due to satisfying these high-level constraints. In contrast, the low-level intermediate representation (IR) in DL compilers exhibits diverse implementations and performs complex hardware-specific optimizations [11], presenting an opportunity to detect more specific, deeper, and severe compiler bugs. Thus, this paper concentrates on the detection of low-level stage bugs to effectively address these challenges.

To the best of our knowledge, there is limited research focused on testing the low-level optimization of DL compilers. One such tool, TVMFuzz [13] generates low-level IRs based on user-defined grammar rules. However, TVMFuzz can only detect bugs with obvious error characteristics, such as crashes, and may not capture more subtle issues like logic errors. Another tool called Tzer [14] performs joint IR-Pass mutations, and compares the results between optimization passes and without them to detect bugs. However, it falls short in detecting non-optimization differential errors. Many bugs, including non-optimization code logic errors, stem from basic code logic mistakes and are not directly related to optimization [4]. These bugs often manifest with symptoms like wrong code behavior, which does not exhibit obvious errors like crashes, yet their impact can be severe [4]. Addressing these categories of bugs is crucial for enhancing the reliability and performance of DL compilers, but unfortunately, existing solutions are currently unable to address these challenges.

Despite the inherent difficulties, it is crucial to acknowledge that the architecture of compilers remains unstable due to the rapid advancements in both compilers and deep learning technologies, driven by the need to meet hardware and algorithmic requirements. Numerous differences exist between various compiler versions, encompassing both optimization and non-optimization code logic disparities. For instance, between November 2021 and April 2023, TVM underwent

2693-9177/23/$31.00 ©2023 IEEE
DOI 10.1109/QRS60937.2023.00066

four versions and had seven development branches, with each update containing at least 7000 commits. Conversely, frequent updates to DL compilers also make the code susceptible to triggering errors. A study shows that 77% of 23k bugs are regressions [15], and the code that has changed recently or frequently is more likely to produce new errors [15]. However, different compiler versions could also be used for differential testing. We can conduct comparisons between executions with different compiler versions to reveal unexpected behaviors and detect errors. If the execution differences for the same models are substantial, these execution models can serve as valuable test seeds, and we can utilize them for effective fuzz testing by mutation.

Thus, in this paper, we propose DeepDiffer, a differential fuzzing approach to maximize the vulnerable differences in different versions of compilers and compare the execution results to effectively detect various bugs in TVM. There are three challenges we need to resolve.

Different compiler versions may be incompatible. With the update of DL compilers, the architecture can sometimes differ, leading to incompatibility issues. There are two kinds of incompatible errors when we aim to ensure the differential testing uses the same inputs. First, TVM may encounter incompatible errors with functions or fields. To address this problem, we strive to fulfill their requirements. For instance, the parsing functions cannot handle IR with different fields so we select the latest versions that have the same fields, which can be parsed by functions. Second, some objects such as `PassNode` and terminal may be bound to the local environment. We cannot run two versions of TVM on one terminal simultaneously and these objects defined in one terminal encounter problems in another terminal. Therefore, we have designed a client-server framework and employed sockets to transmit the IR-Pass pair to the respective terminals. For those objects, we export their identifications (i.e., names) and send them to different terminals. Then we utilize functions to recover and define them within the individual local terminals.

What kind of information should we collect? There are many indicators of program execution, and our goal is to effectively reveal asymmetries. Therefore we choose information pertinent to valuable differences. In the case of DL compilers, the most crucial function is to enhance the runtime performance of DL models. Large time differences imply that some problems lead to the poor performance of the compilers. Based on this observation, we collect information on DL compiler's differential execution paths and time.

To achieve this, we design an instrumentation tool *memdif*, which uses an individual bitmap to collect bitmap information for each iteration. The *memdif* instruments the program (i.e., TVM), and when the execution enters one edge, the corresponding position in the bitmap is set to True. We compress the bitmap where one bit represented a byte and design a function to calculate the number of different bits between bitmaps.

How do we measure the value of executions? Besides collecting information, we require a standard to assess the value of executions for efficient bug detection. Given the abundance of differential executions, we employ priority-guided heuristics to determine their value and apply a strict standard for the seed schedule. As both collected time and execution paths hold equal importance, we propose the use of *diffit* (differential fitness) and two hyperparameters to balance the weights, thereby exposing more differential execution paths and time differences. In each iteration, we calculate the *diffit* to determine the value of execution and maximize its value. To enhance efficiency, we have designed a roulette wheel algorithm to select higher-priority seeds.

To demonstrate the effectiveness of DeepDiffer, we conduct evaluations using the same benchmarks with TVMFuzz and Tzer. The experimental results clearly indicate that our approach achieves higher coverage and exposes a greater number of bugs compared to the other fuzzers. In total, we discovered 12 bugs belonging to 9 different categories of root causes, surpassing the performance of existing fuzzers. The bugs we identified exhibit diverse error characteristics, emphasizing the significance of bug detection in DL compilers.

In summary, the primary contributions of this work are as follows:

- We propose the first priority-guided differential testing framework for DL compiler testing. We propose a new type of DL compiler coverage to maximize the difference and a new metric to determine seed selection and schedule.
- We have compared DeepDiffer against existing low-level IR fuzzers for testing TVM, the experimental results show that DeepDiffer outperforms existing fuzzers with higher coverage and more bugs. DeepDiffer found 13 bugs in TVM with 9 root causes, including 9 bugs that were first detected, and we have reported our findings to the developers of TVM. DeepDiffer has been open-sourced at: https://github.com/KuiliangL/DeepDiffer.

2. Background

2.1. Deep Learning Compilers

Deep learning (DL) has emerged as an effective approach to solving problems in various domains, such as autonomous driving cars [16], software engineering [17], [18], and health care [19], [20]. Several deep learning frameworks including TensorFlow [21], Keras [22], and PyTorch [23], have been developed to facilitate the implementation of DL models. Moreover, various kinds of hardware like NNP [24] have been designed to accelerate the execution of these models.

To achieve this acceleration, DL compilers are designed to take a DL model as input and generate hardware-optimized code as the output for execution on the deployed hardware. DL compilers also leverage third-party mature tool-chains to improve portability. Fig.1 illustrates the general architecture of the DL compiler, which comprises the following stages.

Model Loading: A stage is responsible for loading a DL model and transforming it into a computation graph representation (i.e., high-level IR). The purpose of the high-level IR is to construct the control flow and the dependency between data and operators, while also providing an interface for graph-level

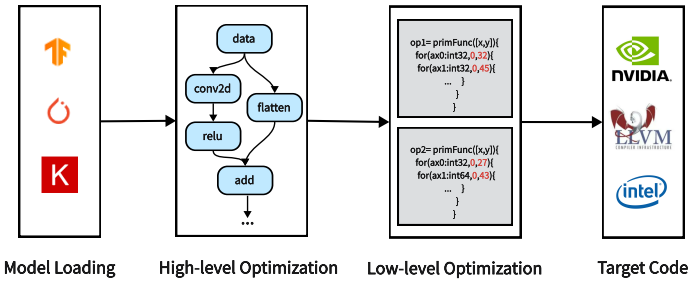

Figure 1. Architecture of DL compilers

optimizations [11]. Each node in the computational graph is represented by one or several IR expressions.

High-Level IR Transformation: A stage responsible for conducting hardware-independent optimizations on high-level IR to reduce redundancy and improve efficiency. DL compilers optimize the computational graph represented in high-level IR and generate optimized IR for further optimization.

Low-Level IR Transformation: A stage that conducts hardware-specific optimization is responsible for improving efficiency and generates optimized code for target hardware. This stage's optimizations can include hardware intrinsic mapping, memory allocation, latency hiding, loop-related optimizations, etc. [25], [11]. It can also directly transform the high-level IR to third-party tool-chains for optimizations [11].

2.2. Fuzzing

Fuzzing [26] is a widely used technique for detecting software bugs and vulnerabilities. The primary concept behind fuzzing is to generate random inputs and test programs to explore unexpected behaviors. Many fuzzers [27], [28], [29] use code coverage as a metric to evaluate fuzzy processes. The fundamental assumption of using coverage is that finding more execution states (e.g., new coverage) will increase the probability of detecting bugs [30]. For example, AFL [31] is a coverage-guided tool that has identified numerous vulnerabilities in diverse applications. Code coverage has been widely recognized as one of the most widely used metrics to evaluate software testing techniques.

2.3. Differential Testing

Differential testing is a mature testing technology for large software systems: a test case is randomly generated, and the output is compared for similar systems. The main goal of differential testing is to identify bugs by observing the asymmetries between different implementations of the same functionality when provided with the same input. There are three widely-used differential-testing strategies in compiler testing [32]:

Cross-compiler strategy: This strategy compares results by different compilers to detect compiler bugs. For example, RandIR [33] uses random instances of the given IR as inputs to detect Scala bugs via cross-compiler differential testing.

Cross-optimization strategy: This strategy compares results by using different optimizations implemented in a single compiler to detect compiler bugs. For example, Sassa and Sudosa [34] use the cross-optimization strategy and detect compiler bugs by comparing traces of important values before and after the program optimization.

Cross-version strategy: This strategy compares results produced by different versions of a single compiler to detect compiler bugs. Chen et al. [35] propose to test JVM implementations via differential testing, focusing on the startup processes of JVM. Their approach uses a cross-compiler strategy and cross-version strategy to detect JVM discrepancies.

2.4. Mutation Strategies

The main idea of program mutation is to modify parts of an existing test program to generate a new test program. In our mutation strategies, we draw inspiration from Tzer's design. We believe that Tzer's mutation strategies are highly representative, and their experiments also demonstrate the superiority of their work. There are three types of strategies: general mutation, domain-specific mutation, and pass mutation.

For the IR mutation, constraints are introduced for mutations to ensure syntactic correctness and mitigate semantic errors. Generating invalid programs might not be very useful for testing compilers since a program undergoes multiple processing stages within the compiler. if a compiler is presented with an invalid input program, then the program tends to get discarded in the initial stages of the processing [32]. We use 3 general mutation strategies: Insertion, Deletion, and Flip. For the domain-specific mutation strategies, we choose several DL-specific Python APIs and mutate them, such as loop nesting, memory operation, and assertion operation. DeepDiffer mutates the pass consequences by combining the pass consequences.

```
inline TryConstFold<tir::Sub>(PrimExpr a, PrimExpr b){
  if(pa && pb){
    int64_t res = pa->value - pb->value;
    return IntImm(rtype,GetFoldResultInt64Repr(res,rtype))
  }
  ...
}
inline int64_t GetFoldResultInt64Repr(int64_t x, const
    DataType& dtype) {
  if (dtype.bits() < 64) {
    x &= (1LL << dtype.bits()) - 1;
  }
  ...
  return x
}
```

Listing 1. Motivation example of version 1

```
IntImm::IntImm(DataType dtype, int64_t value, Span span) {
  ...
  if (dtype.is_uint()) {
    ICHECK_GE(value, 0U);
  }
}
inline TryConstFold<tir::Sub>(PrimExpr a, PrimExpr b){
  ...
  if(pa && pb){
    return IntImm(rtype,pa->value - pb->value);
  }
  ...
}
```

Listing 2. Motivation example of version 2

3. MOTIVATION EXAMPLES

In this section, we illustrate the rationale behind our approach by using a buffer overflow bug example in TVM, which is related to the absence of checks about unsigned 32-bit values.

To trigger this problem, we execute a reduction operator that involves reducing a large unsigned 32-bit integer using a smaller unsigned 32-bit integer. As depicted in Listing 1 and Listing 2, the handling of the reduction operator varies across different versions. TVM defines the `IntImm` type and employs checks on the unsigned integer's value to ensure the accuracy of the interval (line 3, line 4 in Listing 2). Listing 1 represents the source code of the latest version 1. In this version, when TVM handles the reduction operator with an unsigned 32-bit integer parameter, it executes the `GetFoldResultDoubleRepr` function (line 4). In this function, TVM checks the dtype's bit number (line 9). As our parameter is a 32-bit integer, TVM converts the parameter x, which is the reduction result (line 10). However, TVM does not validate the parameter's value, leading to a buffer overflow and returning a wrong value. The compiler returned the value of the reduction result plus `INT_MAX`. Listing 2 shows the early version 2. In this version, TVM performs the reduction calculation and passes the result to the IntImm function (line 10). Then TVM detects the error and throws the exception since the result is a negative number, violating the definition of an unsigned integer (line 3, line 4).

Due to the incorrect arithmetic value returned by this error, it did not exhibit obvious characteristics and was not directly related to the optimization logic. As a result, TVM did not trigger any exceptions and even when executed at different optimization levels, no differences were revealed. Consequently, this incorrectly compiled model went unnoticed by the existing work Tzer's oracles, and the existing fuzzers were unable to detect this wrong code bug. In our paper, we identify this bug by comparing the compilation results obtained from two different compilers.

If we want to effectively detect this category of bugs, there are two challenges. First, the approach must detect the error characteristic of bugs, although some bugs are not caught by TVM and do not reveal differences even in the different optimization levels. Second, the approach must select seeds that have a high quality of revealing error characteristics. In this paper, we design and implement DeepDiffer to solve these challenges.

4. APPROACH

In this section, we introduce the conceptual framework of DeepDiffer, a priority-guided DL compiler fuzzer for differential testing. DeepDiffer conducts differential testing of different versions of TVM compilers. The illustrative depiction of DeepDiffer's architecture is presented in Fig.2. DeepDiffer stands as a client-server framework that compiles the IR file in parallel. The client has a seed pool, mutators, oracles, and other components. The server shoulders the responsibility of execution, which is used to receive the messages of the IR-Pass pair, compile the seed, build the model and transmit results to the client.

DeepDiffer first selects an IR-Pass pair from the seed pool (①). The chosen seed undergoes mutation, resulting in the creation of a novel mutated seed (②). The mutated seed is subsequently dispatched to the different TVM compilers for compilation and execution (③). During compilation and transmission, DeepDiffer conducts differential testing by compiling the same seed with different DL compilers. It collects and compares differential messages, including the compilation results, build time, and the differential coverage branches (④). Based on the collected information, DeepDiffer derives *diffit* (differential fitness) (⑤). Our goal is to maximize *diffit* aiming to pinpoint vulnerable seeds. Collected messages aid in deciding whether the mutated seed should be put into the seed pool as determined by a comparison of the *diffit* (⑥). We set two controls for limiting the mutations. Seeds that do not increase differences many times have their *diffit* reset to the initial priority, permitting further file mutations (①). Test oracles are employed to detect potential bugs (⑦). Input pairs that contravene test oracles are identified and reported for debugging purposes (⑧). Algorithm 1 shows the details of the fuzzing loop, and we will further explain it in the following subsections.

Algorithm 1 Fuzzing Loop

Input: Set of initial seed S_0, budget time T, pass control Q_{max}, IR control I_{max}

```
1:  S ← S_0
2:  while time budget T do
3:      (F, P, D, I, Q) ← RouletteWheelSelect(S)
4:      F' ← MutateIR(F)
5:      diffinfo_1 ← RunTVM_1(F', P)
6:      diffinfo_2 ← RunTVM_2(F', P)
7:      D' ← CalcDiff(diffinfo_1, diffinfo_2)
8:      if ∃error then
9:          Report(F', P)
10:     else if D' > D then
11:         S ← S ∪ (F', P, D', 0, 0)
12:         S.update(F, P, D, 0, 0)
13:     else
14:         S.update(F, P, D, I + 1, Q)
15:     if I ≥ I_max and Q < Q_max then
16:         P' ← MutatePass(P)
17:         diffinfo_1 ← RunTVM_1(F, P')
18:         diffinfo_2 ← RunTVM_2(F, P')
19:         D' ← CalcDiff(diffinfo_1, diffinfo_2)
20:         if ∃error then
21:             Report(F, P')
22:         else if D' > D then
23:             S.update(F, P', D', 0, 0)
24:         else
25:             S.update(F, P, D, I, Q + 1)
26:     if I ≥ I_max and Q ≥ Q_max then
27:         S.update(F, P, D_0, 0, 0)
```

4.1. Seed Schedule

Regarding the seed pool, each input for DeepDiffer is represented by a tuple $\langle F, P, D, I, Q\rangle$, where F represents

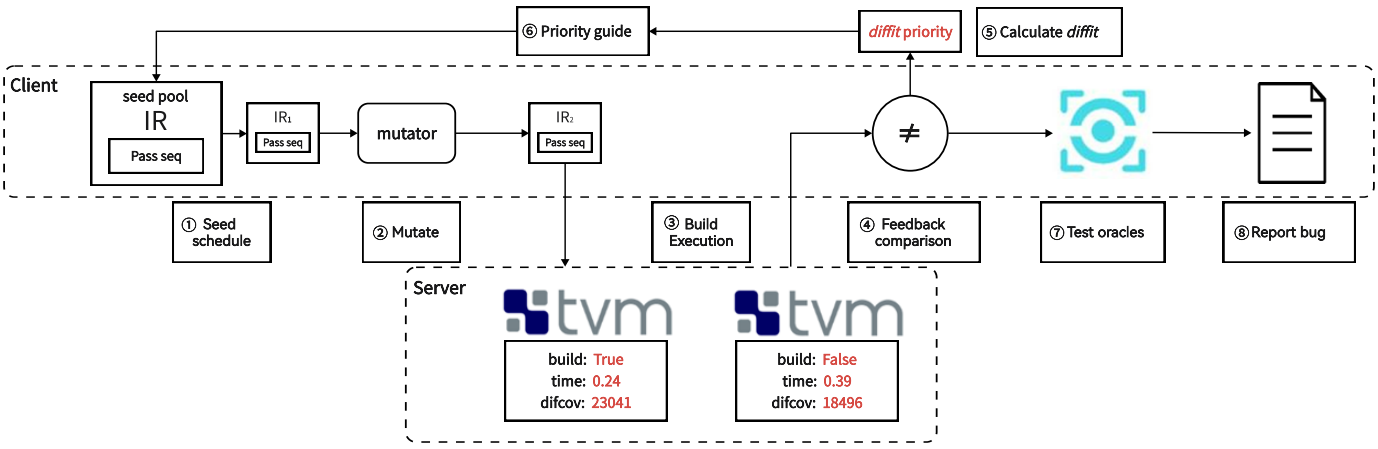

Figure 2. System Overview

an IR file, P represents the sequence of passes for the IR, D represents the *diffit* value, and two controls I, Q are employed for managing the IR and pass mutations. DeepDiffer first initializes the seed pool with I and Q set to zero, and initial *diffit* D_0=1 for all seeds. In each iteration, DeepDiffer selects a high-priority seed and avoids local optimum utilizing the priority-guided roulette wheel algorithm (① in Fig.2). Algorithm 2 delineates the intricacies of the roulette wheel algorithm. The process commences by summing the *diffit* values, followed by the random generation of a probability denoted as p. Subsequently, the probability of each seed is computed, and a comparison is drawn to calculate the index of the chosen seed. Once chosen, the seed is subjected to mutation through the random selection of a mutator. DeepDiffer uses sockets to transmit the seed pair and the seed is then compiled and executed in different TVM compilers (② , ③ in Fig.2).

For each IR file, successful compilation in both versions is a prerequisite. Furthermore, if the mutation results in an increased differential, we retain the mutated seed to control the seed pool's size because of the numerous differences. We then reset the control value of the selected seed to zero since this mutation is rewarding. If this mutation fails to augment the differential, we increment the corresponding control value. DeepDiffer also seeks a better P' to pair with F upon reaching the threshold value of I_{max}. If a seed fails to increase the *diffit* after undergoing mutations for I_{max} and Q_{max} cycles, the *diffit* value is reset to its initial state, and the control value is reset to zero, facilitating further file mutations and diminishing the current seed's priority (⑥ , ① in Fig.2).

Algorithm 2 Roulette Wheel Algorithm

Input: Set of initial seed S_0
Output: the index of the seed and the chosen seed

$S \leftarrow S_0$
$p \leftarrow$ RANDOM(0, 1)
$sum \leftarrow \sum_i S_i(D)$
$r \leftarrow 0$, $idx \leftarrow 0$
while $r \leq p$ **do**
    $r \leftarrow r + \frac{S_{idx}(D)}{sum}$
    $idx \leftarrow idx + 1$
return idx, S_{idx}

4.2. Comparison and *Diffit*

In this paper, we undertake a comparative analysis of different compiler versions. We subject the compilers to the same seeds for compilation, then evaluate and contrast the outcomes, including build time and coverage branches aiming to gather differential information pertaining to these compilers (④ in Fig.2). To facilitate this, we introduce a coverage instrumentation tool named *memdif*, which is an extension of LLVM's Sanitizer Coverage and *memcov* [14]. The *memdif* employs an individual bitmap to capture coverage branches for each iteration thereby enabling an approximate calculation of differential coverage for the DL compiler. This "DL compiler differential coverage" is branches that exhibit uniqueness in the different versions. We approach this feedback as an optimization problem and introduce the concept of the *diffit*:

$$\textit{diffit} = a * \frac{cov_1 \cup cov_2 - cov_1 \cap cov_2}{cov_1 \cup cov_2} + b * |t_1 - t_2| \quad (1)$$

The hyperparameters a and b serve the purpose of balancing the two objectives. The first part indicates the ratio of DL compiler differential coverage to the combined total of branches. This quantification assesses differences in collected coverage data. Given the structural similarity between the two chosen versions of TVM, we infer a corresponding similarity in their gathered coverage data. Therefore we can compare their program execution paths. Distinct execution paths potentially reflect varying code logic or structural updates, potentially revealing and pinpointing prospective errors. The subsequent part represents the temporal discrepancy observed during the compilation of the two compiler versions. Compilation time stands as a pivotal gauge of compiler efficiency. Thus, we employ these factors to construct *diffit*. The *diffit* is applied to the quantification of the seed values thereby informing our determination of seed selection and schedule. In our experiment, the majority of the *diffit* distribution ranges from 1 to 10.

4.3. Oracle Handling

Test oracle serves as a crucial mechanism for assessing the success or failure of a test. In this paper, we encompass

a total of four test oracles. These oracles draw inspiration from Tzer's design, which we have modified in accordance with our comparative analysis. By comparison, any test case that violates test oracles is reported and the reproduction and localization of bugs are achieved through manual analysis. (⑧ in Fig.2).

Result Inconsistency: DeepDiffer conducts comparisons between two versions and different optimization levels. For each generated input file, DeepDiffer submits it to the distinct DL compilers for compilation and compares their results. An inconsistency bug is identified if the absolute or relative error surpasses the predefined threshold.

Performance Inconsistency: DeepDiffer assesses timing disparities between two versions and different optimization levels. It quantifies the build time for each compiler version. We set up a defined performance threshold. If the time difference surpasses this margin or the optimized IR has a slower running speed compared to the unoptimized IR, it is identified as a potential performance error.

Dead Loop Problem: While in the compilation phase, DL compilers might continue running without properly terminating or producing outcomes. To circumvent the possibility of the fuzzing process becoming stuck or consuming an excessive time budget, DeepDiffer establishes a maximum time threshold. Should the compiler's runtime exceed this threshold, it is identified as a dead loop error.

Runtime Failure: Crashes or unexpected exceptions represent errors requiring Oracle intervention, which we classify as runtime failures. This error is identified by the examination of `exitcode`.

5. Implementation

Due to code modifications and environment dependencies, incompatible errors arose during our architecture implementation. Running TVM requires setting the Python environment `PYTHONPATH` to specify the library's location. As a result, the terminal continuously runs TVM with the bound version and could not switch to another version until program termination. To overcome this limitation, we employ a client-server framework, enabling parallel fuzzing to enhance efficiency. Furthermore, we selected two of the latest versions with identical fields as TVM employed `JSONReader` and related functions to parse IR files and these two latest versions had more undiscovered potential errors. Parsing the IR with fields not recognizable by `JSONReader` triggers exceptions that TVM could not find related fields. During the transmission, we convey the names of `PassNode` and reconstruct them in server terminals to address "bad function call" errors in pass transformations, which could occur if the `PassNode` was not localized within the environment. In addition, we reproduce the passes using recorded functions manually because the maintained passes have similar errors as well.

The seed mutation of DeepDiffer is implemented based on Tzer's framework. It incorporates three types of mutators, including 3 general-purpose mutators, several domain-specific mutators, and several pass mutators. Regarding the strategies for IR mutation, we utilize TVM operator APIs to create sub-expressions that meet specified constraints. Additionally, we modify the consequences of passes to mutate passes.

To transmit IR-Pass pairs to TVM, DeepDiffer employs a client-server framework and utilizes Python's socket functionality for message transmission. Different versions of TVM are configured as servers, while other components like the seed pool and mutator function as clients. The client selects a seed, applies mutations, and forwards messages to the servers. The servers compile the data and gather information, which is subsequently sent back to the client for comparative analysis and bug report. We execute comparisons with the TVM v0.10-dev (ee319d9) version and TVM v0.9 (d361585) version, designing TVMv0.10-dev as the master version. The fuzzing loop operates within a sub-process, continuing until the allocated time budget is exhausted.

Based on *memcov* [14], we design and update the coverage collector tools, *memdif*. This involves the maintenance of two distinct bitmaps: an overall bitmap is devised to collect the overall coverage information, while an individual differential bitmap is engineered to collect differential information. We compress the bitmaps whose size is the number of edges in TVM divided by eight (byte size). we employ a Python interface to retrieve the coverage state.

Once a test file violates the test oracles, the reporter records essential data required for reproducing the failure, including the selected seed, the mutated seed, IR, passes, and the bug classifications defined manually. Upon replicating the potential issue, tests are conducted across four versions: TVM v0.10-dev, TVM v0.9, TVM v0.11-dev (5019dce) and TVM v0.9-dev (8f6543e). In scenarios involving seed usage, DeepDiffer deploys over 600 TIR functions derived from `tvm.relay.testing`. If seeds are not employed, DeepDiffer undertakes mutation with an empty function.

6. Experiment

6.1. Metrics

Our evaluation primarily focuses on the following metrics:

Coverage: Following prior studies [36], [14], [37], we trace source-level branch coverage within the test files, measuring total coverage counts of all hit branches. To ensure equitable comparison, our counting exclusively occurs during the compilation and execution phases, disregarding coverage during the initialization phase. Furthermore, we enumerate differential unique coverage branches that represent those branches that remain uncovered by other compilers.

Bug Counting: Following prior studies [38], [14], [4], we employ the count of distinct patches as an indicator of the number of identified bugs.

6.2. Baselines

In order to assess the effectiveness of DeepDiffer, we conduct a comparative analysis with fuzzers detecting low-level IR bugs.

TVMFuzz: TVMFuzz is the first generation-based low-level IR fuzzer targeting TVM. Its objective is to autonomously generate various low-level IRs utilizing a user-defined probability table for fuzzing TVM.

Tzer: Tzer is a coverage-guided and mutation-based fuzzer tailored for TVM's low-level IR. Tzer leverages coverage feedback to perform joint IR-Pass mutations concentrating on identifying bugs within low-level optimizations.

6.3. Experimental Configuration

The testbed hardware configurations include 1) Intel E5-2650 CPU (40 threads); 2) 125 GB memory; and 3) 2TB NVMe SSD. The operating system employed is Ubuntu18.04 and targeted DL systems are compiled using Clang 11.1.0 under release mode. The default software versions used in the evaluation are TVM v0.10-dev and TVM v0.9.

In our evaluation, we focus on the LLVM target with TVM being compiled under level-2 compilation. For the sake of achieving stable coverage trends, we conduct five iterations of experiments and subsequently average the resulting data. Following prior research [14], we gather coverage data for all compared techniques within the default 4-hour time allocation using *memdif*. We set D_0 to 1, I_{max} to 6 and Q_{max} to 1 by default. For the RQ1 experiment, both Tzer and TVMFuzz are tested using the TVM 0.10-dev version. Both Tzer and DeepDiffer are configured with default settings including the same seeds. We employ the differential test mode with Tzer for the experiments, comparing distinct optimization outcomes once per iteration. Since TVMFuzz does not require coverage feedback, we execute non-instrumented TVM for 4 hours, collecting the generated IR files. Then these files are compiled within the instrumented TVM to collect coverage.

6.4. Research Questions

We study the following questions to evaluate DeepDiffer:

- RQ1: How is the effectiveness of DeepDiffer compared with other fuzzing techniques in testing TVM?
- RQ2: How do different parameter settings and experimental setups impact DeepDiffer's effectiveness?
- RQ3: Does priority feedback contribute to the effectiveness of DL compiler fuzzing??
- RQ4: How effective is DeepDiffer in detecting bugs?

7. Evaluation

7.1. Coverage Efficiency

Initially, we assess the coverage trends of both DeepDiffer and compared existing methods to ascertain whether DeepDiffer attains superior coverage and enhanced efficiency. Fig.3 illustrates the coverage trends within the default 4-hour budget. Fig.4 shows the coverage trends with the generation files over the same four hours.

Fig.3 illustrates that DeepDiffer exhibits a coverage trend similar to Tzer while surpassing Tzer in terms of coverage counts. Both Tzer and DeepDiffer achieve higher coverage compared to TVMFuzz. Since Tzer and DeepDiffer employ the same seed pool and mutation strategies, their search space

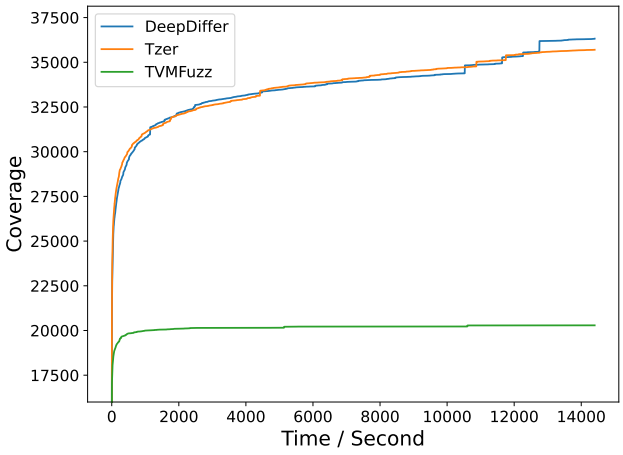

Figure 3. Comparison of time efficiency with existing work

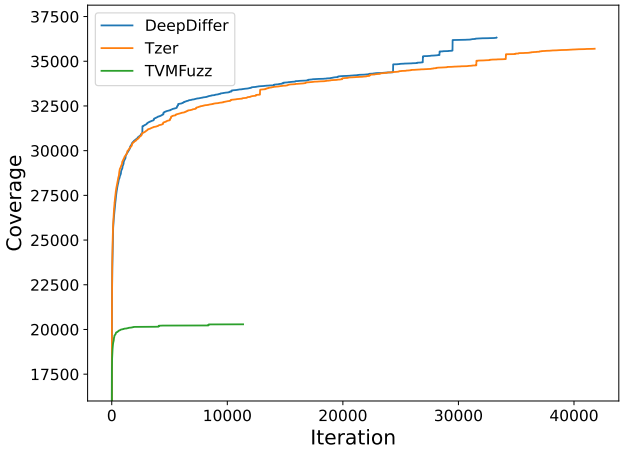

Figure 4. Comparison of iteration efficiency with existing work

is identical. This observation suggests that even though DeepDiffer does not directly utilize coverage feedback, priority-guided feedback can enhance coverage. The coverage trends between Tzer and DeepDiffer appear alike, yet DeepDiffer's trend experiences noticeable spikes in certain iterations. This behavior might be attributed to the fact that the *diffit* metric is not directly linked to coverage.

Fig.4 reveals that Tzer exhibits a quicker runtime compared to DeepDiffer and TVMFuzz (the x-axis presents the iteration). DeepDiffer employs the client-server framework to facilitate message transmission and calculates the *diffit*, leading to a relatively slower execution. This observation underscores DeepDiffer's capability to generate higher-quality tests.

Additionally, we utilize a Venn diagram to visualize the distribution of coverage branches. Fig.5 reveals that DeepDiffer encompasses 2221 unique branches, while Tzer and TVMFuzz exhibit 1031 and 896 unique branches. Notably, DeepDiffer's tally of unique branches surpasses Tzer's by nearly 100% and TVMFuzz's by 247%. This observation underscores DeepDiffer's capacity to uncover more unique branches and explore deep paths.

Subsequently, we count the number of bugs and categorize them by bug types. Table I provides a concise overview of the bugs and bug types identified across DeepDiffer and studied

baselines. Notably, DeepDiffer exhibits a higher count of both bugs and distinct bug types. Further elaboration on these findings is presented in section 7.4.

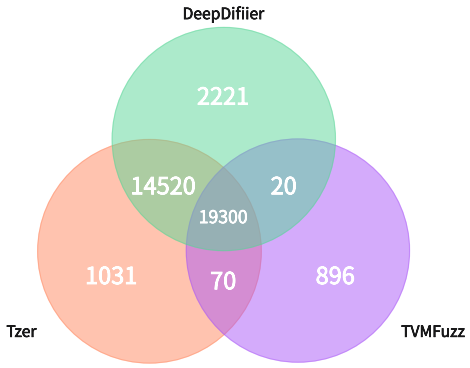

Figure 5. Venn diagram of coverage

TABLE I
BUGS FOUND BY DEEPDIFFER AND EXISTING WORK

	DeepDiffer	Tzer	TVMFuzz
Bugs	**13**	8	4
Bugs Type	**9**	7	4

7.2. Parameter Study

We evaluate the parameter seed, time budget, and control parameters to determine the optimal configuration and test how varying parameter settings impact the effectiveness.

Study on seeds S_0: DeepDiffer has the option to utilize either the initial seed pool or generate input files from an empty function. Fig.6 depicts the performance of DeepDiffer with and without the initial seeds. It is evident that the seed-enabled version outperforms the non-seed version, primarily due to the higher quality of seeds generated when utilizing the seed pool. DeepDiffer within many TIR seeds introduces greater diversity. Conversely, the non-seed DeepDiffer generates seeds iteratively, but their mutations are influenced by early seeds, which may not contribute to the same level of diversity. Therefore, the inclusion of an initial seed pool proves essential for our experiments.

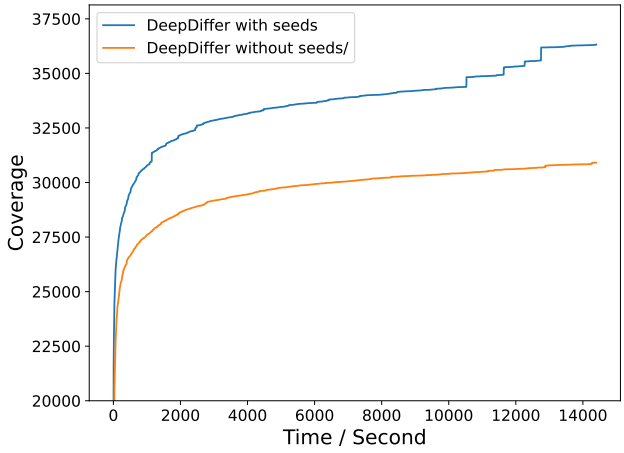

Figure 6. Impact of the seeds on DeepDiffer

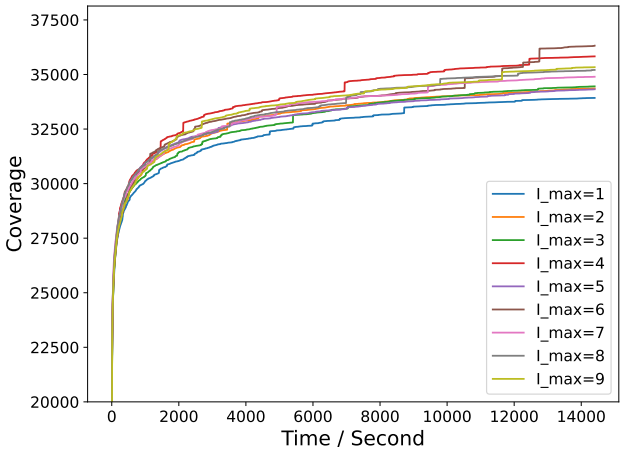

Figure 7. Impact of parameter I_{max} across time

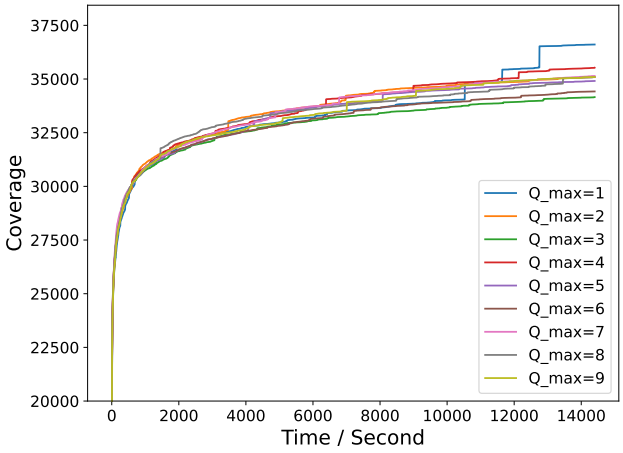

Figure 8. Impact of parameter Q_{max} across time

Study on IR-Pass control: DeepDiffer takes two parameters to control the interleaving of IR-Pass mutation: I_{max} and Q_{max}. To assess the influence of I_{max} and Q_{max}, we conduct the experiments using various values of the I_{max} and Q_{max}. Fig.7 presents detailed coverage trends associated with different I_{max} values. The observations from Fig.7 indicate that I_{max} =6 demonstrates the best performance while I_{max} =1 exhibits the least effective performance. Fig.8 illustrates that Q_{max}=1 demonstrates the best effectiveness, indicating that frequent pass mutations do not significantly enhance coverage. In conclusion, it is crucial to distribute the frequency of IR and pass mutations appropriately.

Study on fuzzing time: Fig.9 depicts the coverage trend over a span of 12 hours. As observed in Fig.9, the initial 3-hour interval accounts for 93.4% coverage and the first 4-hour window contributes 94.8% coverage. This indicates that our framework is equally efficient as Tzer (the first 4-hour window contributes 94.9% coverage) during the early phase. Given the significant contributions observed within the 4-hour window and prior research [14], we select 4 hours as the default time budget.

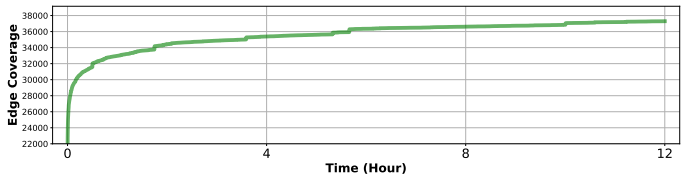

Figure 9. 12-hour coverage trend of DeepDiffer

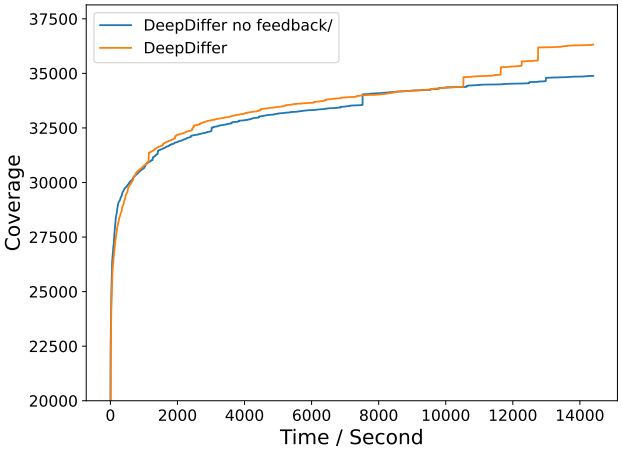

Figure 10. Coverage trends between DeepDiffer and non-DeepDiffer

7.3. Contribution of Priority Feedback

We compare the DeepDiffer with the variant that omits the use of priority feedback and assess their respective coverage to demonstrate the impact of priority feedback. Fig.10 illustrates the coverage trends, enabling us to determine the effectiveness of the *diffit*. As evident from Fig.10, the *diffit* feedback proves valuable in attaining higher coverage. The non-diffit variant maintains seeds without considering the *diffit* comparisons. Consequently, the non-diffit variant accumulates an excessive number of seeds, thereby diminishing runtime efficiency and hampering the generation of higher-quality seeds.

7.4. Bug Root Cause Analysis

To date, DeepDiffer has identified 9 root causes of bugs as detailed in Table II. In the subsequent sections, we will discuss representative bugs from each category and conduct comparisons between bug categories across different baselines.

Type Error: This category of bugs pertains to type-related issues. For example, DeepDiffer identified a mutated file that utilized the `tir.isnullptr` and `tir.call_intrin` functions without verifying the parameter's `dtype`. When a different `dtype` (such as a floating-point type) was supplied as a parameter for compilation, TVM encountered a segmentation fault. This occurred because the `isnullptr` function is intended for integers and does not validate the `dtype`. A mismatch between the provided `dtype` and the expected integer `dtype`, resulted in the crash. Similarly, problems arose with other functions like `tir.isnan` function, which is designed for floating-point operations, and experienced similar issues if an incorrect `dtype` was employed as a parameter.

Memory Error: TVM employs C++ to implement its core components, making memory-related issues quite prevalent. These include problems like invalid memory access and out-of-bounds access. DeepDiffer detected an example where TVM failed to validate `NULL` parameters during the invocation of certain APIs such as `BufferStore(None, None, None)`, resulting in a crash during function execution due to TVM's lack of `NuLL` object handling in many functions or APIs. Consequently, when these functions receive a `NULL` object, a bug is triggered.

Arithmetic Error: This root cause involves incorrect numerical computations, values, or usages. DeepDiffer detected a floating-point exception when the `floordiv` function was employed with a specific value. When we utilized the divide function "x/y" where "x" is an integer and "y" is cast to zero, TVM computed the divide function and returned zero. In the implementation of the divide function, TVM checks the divisor's value to make ensure it's not zero. We configured the function call that returned a boolean parameter as the divisor "y", such as `tir.isnan` function. If this function returned "True" and we cast the parameter's `dtype` and converted it to the integer `dtype`, the convention value became zero. However, within the division function, the zero may be transformed into INT_MAX, leading to an incorrect result. We speculate that certain APIs are sensitive to the expression or value, which may trigger this issue.

API Misuse: This category of bugs arises from misunderstandings of APIs. For example, DeepDiffer discovered a bug where execution ran significantly slow due to an unroll `For` statement with an extensive loop length, resulting in a dead loop error according to the oracle. When we tested the same test file with alternative loop attribute keys (e.g., parallel), the execution was swift. Thus, we infer that TVM failed to optimize the large value effectively using the unroll attribute.

API Inconsistency: When an API deviates from the expected behavior, it may indicate API inconsistency. A bug was identified by DeepDiffer involving the usage of `tir.isfinite` alongside the `tir.call_intrin` within a `For` statement, which resulted in a crash error. Further investigation revealed that the `tir.isinf` and `tir.isfinite` functions were designed to be composed of `tir.abs` and `tir.isnan` functions. Other functions were formulated to implement the corresponding APIs. The issue arose from an inconsistency in the function sizes, with `tir.infinite` having an additional size of 2, while the size of the referenced called function was 1. This inconsistency led to a crash scenario.

Exception Handle: This category of bugs occurs when the compiler does not provide correct or precise error messages, and sometimes fails to throw an exception when it should or not should. DeepDiffer found a bug when using certain functions, such as the `tir.exp` and `tir.log`. When we provided an integer variable as input, TVM did not find the corresponding intrinsic declaration. This discrepancy arises because some functions are designed for floating-point while some functions are designed for integers. TVM does not design the exception to handle wrong, leading to unclear error messages. In the case of buffer flow errors, TVM does not validate

TABLE II
SUMMARY OF BUGS FOUND BY DEEPDIFFER AND BASELINES

Route Cause	Description	DeepDiffer	Tzer	TVMFuzz
Type	Type mismatch with functions	✓	✓	
Memory	Crash when the parameter is NULL	✓	✓	✓
AE	Cast zero in the divisor convert to another value	✓		
AE	Float point error when big integer divide -1	✓	✓	✓
API misuse	Bad performance when using the specific loop interval	✓		
API misuse	Bad performance when using the `Unroll` loop type	✓	✓	
API misuse	Bad performance when using the `UnrollLoop` pass	✓	✓	
API-I	Inconsistency with the size when calling the `tir.isfinite`	✓	✓	
Exception handle	Wrong error messages when using the TIR function	✓	✓	✓
Buffer overflow	Overflow when using the reduction that results in a negative unsigned 32-bit value	✓		
Buffer overflow	Overflow when claiming a unsigned 32-bit const integer	✓		
Code logic	Different results when conducting the attribute and IfThenElse statement	✓		
Assignment Error	Crash if the variable is not certain in `LetStmt`	✓	✓	✓

AE: Arithmetic Error; API-I: API Inconsistency

the `dtype` when casting unsigned integer values, resulting in buffer overflow errors. This issue could be mitigated by implementing additional checks.

Buffer Overflow: DeepDiffer identified two buffer overflow errors within TVM. In addition to the motivation example, another scenario arises when declaring an unsigned constant value. TVM utilizes the `tir.const` function to declare constants by invoking `MakeConstScalar` function. During this process, TVM examines the type of the parameter. If the parameter is unsigned, TVM casts it to an unsigned 64-bit type. If the cast value surpasses the `INT_MAX`, TVM then converts the value to a 32-bit type. In a specific case where an unsigned 32-bit integer negative constant value is claimed, TVM matches the `dtype` and performs the cast operation, but neglects to verify the validity of the unsigned int value. This results in a buffer overflow occurrence. It is the user's responsibility to avoid declaring negative unsigned integers but we suggest that we can add some assertions to identify these problems.

Code Logic Error: Logic errors are issues within a program that cause it to deviate from its intended purpose. DeepDiffer detected a code logic error during the execution of an if-then statement. This statement encompassed a condition and two statements, `AttrStmt` and `IfThenElse` statement. Upon investigation, we observed that in cases where the condition evaluates to false, the associated `AttrStmt` statement would not be checked as intended. We devised an expression involving the reduction of an unsigned 32-bit integer value by a larger unsigned 32-bit integer value or a division by zero operation as a parameter of `AttrStmt`. In the TVM 0.10-dev version, the attribute statement goes unchecked, leading to the absence of an error trigger. However, in the TVM 0.9-dev version, careful scrutiny of the two statements results in error detection. The reason behind this discrepancy lies in TVM's behavior of omitting attribute value checks in some conditions.

Assignment Error: DeepDiffer has identified a bug stemming from situations where variables lack proper initialization or are not correctly assigned or bound. Consider the case of "let v= v$\geq$0?1:2". In this scenario, the variable "v" is associated with a condition that determines its value. Different conditions yield distinct values, yet TVM failed to interpret the condition due to the absence of a specific value for the variable. This led to TVM throwing a segmentation error.

8. RELATED WORK

Traditional Compiler Testing: Numerous traditional techniques have been developed and have achieved significant success in many domains. For instance, AFL [31] and LibFuzzer [10] are widely-used binary fuzzers that incorporate CFG tools to detect bugs across various applications. These tools primarily take programs as inputs and do not account for DL optimizations. As a result, they encounter challenges when parsing DL models.

DL Compiler Testing: Several works are dedicated to detecting bugs at the high-level stage. MT-DLComp [39] leverages metamorphic testing and mutates the existing models to identify bugs. HirGen [12] and NNSmith [40] focus on model generation techniques, extracting high-level IR constraints to guide model generation. they add or replace a new node in the computation graph and verify whether operators violate constraints. TVMfuzz [4] learns from the API call dependencies and mutates the API to create new unit tests.

Although certain testing techniques for DL frameworks are not specifically designed to detect compiler bugs, they can still be applied to detect compiler bugs, particularly at the high-level stage. For instance, LEMON [41] tests DL frameworks by mutating Keras models. Some techniques target fuzzing DL operators but they may struggle to detect bugs in high-level optimizations [40], such as FreeFuzz [42], ACETest [43], etc.

Some works focus on detecting bugs in the low-level stage. TVMFuzz [13] is the first fuzzer targeting TVM. TVMFuzz generates TIR expressions randomly based on user-defined grammar rules. Tzer [14], on the other hand, utilizes coverage

feedback and conducts IR-Pass mutation to detect bugs in TVM.

9. CONCLUSION

DL compiler bugs exhibit a wide range of error characteristics, some of which are challenging to identify and can lead to significant consequences. In response, we have developed a priority-guided differential testing framework for TVM. We present DeepDiffer, a testing framework based on Tzer, designed to target low-level IR defects within the TVM compiler. The evaluation demonstrates that DeepDiffer outperforms other Low-level IR fuzzers, Tzer and TVMFuzz. To date, we have identified 13 bugs with 9 root causes, and 9 of these bugs were detected for the first time. Our findings have been confirmed by the TVM community's developers and we hope our work will continue to enhance the reliability of the DL compilers.

ACKNOWLEDGMENT

This work was supported by National Natural Science Foundation of China under Grant No. 62372268, Shandong Provincial Natural Science Foundation (No. ZR2020MF055, No. ZR2021LZH007, No. ZR2022LZH013, No. ZR2020QF045), Jinan City "20 New Universities" Funding Project (2021GXRC084), the Zhejiang Provincial Natural Science Foundation of China under Grant No. LY22F020022, and the National Natural Science Foundation of China under Grant No. 61902098.

REFERENCES

[1] Tvm. https://tvm.apache.org/, 2021.

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

[3] Wei Liu, Haikun Liu, Xiaofei Liao, Hai Jin, and Yu Zhang. Ngraph: Parallel graph processing in hybrid memory systems. *IEEE Access*, 7:103517–103529, 2019.

[4] Qingchao Shen, Haoyang Ma, Junjie Chen, Yongqiang Tian, Shing-Chi Cheung, and Xiang Chen. A comprehensive study of deep learning compiler bugs. In *Proceedings of the 29th ACM Joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pages 968–980, 2021.

[5] Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, Dingyi Fang, Xiaoyang Sun, Lizhong Bian, Haibo Wang, and Zheng Wang. Automated conformance testing for javascript engines via deep compiler fuzzing. In *Proceedings of the 42nd ACM SIGPLAN international conference on programming language design and implementation*, pages 435–450, 2021.

[6] Shafiul Azam Chowdhury, Sohil Lal Shrestha, Taylor T Johnson, and Christoph Csallner. Slemi: Equivalence modulo input (emi) based mutation of cps models for finding compiler bugs in simulink. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 335–346, 2020.

[7] Junjie Chen, Guancheng Wang, Dan Hao, Yingfei Xiong, Hongyu Zhang, and Lu Zhang. History-guided configuration diversification for compiler test-program generation. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 305–316. IEEE, 2019.

[8] Cen Zhang, Yuekang Li, Hongxu Chen, Xiaoxing Luo, Miaohua Li, Anh Quynh Nguyen, and Yang Liu. Biff: Practical binary fuzzing framework for programs of iot and mobile devices. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1161–1165. IEEE, 2021.

[9] Caroline Lemieux and Koushik Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 475–485, 2018.

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

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

[12] Haoyang Ma, Qingchao Shen, Yongqiang Tian, Junjie Chen, and Shing-Chi Cheung. Fuzzing deep learning compilers with hirgen. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 248–260, 2023.

[13] David Pankratz. Tvmfuzz: Fuzzing tensor-level intermediate representation in tvm. https://github.com/dpankratz/TVMFuzz, 2020.

[14] Jiawei Liu, Yuxiang Wei, Sen Yang, Yinlin Deng, and Lingming Zhang. Coverage-guided tensor compiler fuzzing with joint ir-pass mutation. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA1):1–26, 2022.

[15] Xiaogang Zhu and Marcel Böhme. Regression greybox fuzzing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2169–2182, 2021.

[16] Chenyi Chen, Ari Seff, Alain Kornhauser, and Jianxiong Xiao. Deepdriving: Learning affordance for direct perception in autonomous driving. In *Proceedings of the IEEE international conference on computer vision*, pages 2722–2730, 2015.

[17] Shaohuai Shi, Qiang Wang, Pengfei Xu, and Xiaowen Chu. Benchmarking state-of-the-art deep learning software tools. In *2016 7th International Conference on*

Cloud Computing and Big Data (CCBD), pages 99–104. IEEE, 2016.

[18] Cody Watson, Nathan Cooper, David Nader Palacio, Kevin Moran, and Denys Poshyvanyk. A systematic literature review on the use of deep learning in software engineering research. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(2):1–58, 2022.

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

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

[21] Tensorflow. https://www.tensorflow.org/, 2021.

[22] Keras. https://keras.io/, 2021.

[23] Pytorch. https://pytorch.org/, 2021.

[24] Brian Hickmann, Jieasheng Chen, Michael Rotzin, Andrew Yang, Maciej Urbanski, and Sasikanth Avancha. Intel nervana neural network processor-t (nnp-t) fused floating point many-term dot product. In *2020 IEEE 27th Symposium on Computer Arithmetic (ARITH)*, pages 133–136. IEEE, 2020.

[25] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. Tvm: An automated end-to-end optimizing compiler for deep learning. *arXiv preprint arXiv:1802.04799*, 2018.

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

[27] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1032–1043, 2016.

[28] Qian Yang, J Jenny Li, and David Weiss. A survey of coverage based testing tools. In *Proceedings of the 2006 international workshop on Automation of software test*, pages 99–103, 2006.

[29] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Superion: Grammar-aware greybox fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 724–735. IEEE, 2019.

[30] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. Fuzzing: a survey for roadmap. *ACM Computing Surveys (CSUR)*, 54(11s):1–36, 2022.

[31] American fuzzing lop (afl). https://lcamtuf.coredump.cx/afl/, 2018.

[32] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. A survey of compiler testing. *ACM Computing Surveys (CSUR)*, 53(1):1–36, 2020.

[33] Georg Ofenbeck, Tiark Rompf, and Markus Püschel. Randir: differential testing for embedded compilers. In *Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala*, pages 21–30, 2016.

[34] Masataka Sassa and Daijiro Sudosa. Experience in testing compiler optimizers using comparison checking. In *Software Engineering Research and Practice*, pages 837–843. Citeseer, 2006.

[35] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. Coverage-directed differential testing of jvm implementations. In *proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 85–99, 2016.

[36] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2329–2344, 2017.

[37] Marcel Böhme, László Szekeres, and Jonathan Metzman. On the reliability of coverage-based fuzzer benchmarking. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1621–1633, 2022.

[38] Anjiang Wei, Yinlin Deng, Chenyuan Yang, and Lingming Zhang. Free lunch for testing: Fuzzing deep-learning libraries from open source. In *Proceedings of the 44th International Conference on Software Engineering*, pages 995–1007, 2022.

[39] Dongwei Xiao, Zhibo Liu, Yuanyuan Yuan, Qi Pang, and Shuai Wang. Metamorphic testing of deep learning compilers. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 6(1):1–28, 2022.

[40] Jiawei Liu, Jinkun Lin, Fabian Ruffy, Cheng Tan, Jinyang Li, Aurojit Panda, and Lingming Zhang. Nnsmith: Generating diverse and valid test cases for deep learning compilers. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 530–543, 2023.

[41] Zan Wang, Ming Yan, Junjie Chen, Shuang Liu, and Dongdi Zhang. Deep learning library testing via effective model generation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 788–799, 2020.

[42] Yinlin Deng, Chenyuan Yang, Anjiang Wei, and Lingming Zhang. Fuzzing deep-learning libraries via automated relational api inference. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 44–56, 2022.

[43] Jingyi Shi, Yang Xiao, Yuekang Li, Yeting Li, Dongsong Yu, Chendong Yu, Hui Su, Yufeng Chen, and Wei Huo. Acetest: Automated constraint extraction for testing deep learning operators. *arXiv preprint arXiv:2305.17914*, 2023.