Differential Testing of Cross Deep Learning Framework APIs: Revealing Inconsistencies and Vulnerabilities

Zizhuang Deng[1,2], Guozhu Meng[1,2,*], Kai Chen[1,2,*], Tong Liu[1,2], Lu Xiang[1,2], Chunyang Chen[3]
[1]*SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences, China*
[2]*School of Cyber Security, University of Chinese Academy of Sciences, China*
[3]*Monash University, Australia*
{dengzizhuang, mengguozhu, chenkai, xianglu}@iie.ac.cn, liutong@shanghaitech.edu.cn, chunyang.chen@monash.edu

Abstract

With the increasing adoption of deep learning (DL) in various applications, developers often reuse models by, for example, performing model conversion among frameworks to raise productivity. However, security bugs in model conversion may make models behave differently across DL frameworks, and cause unpredictable errors. Prior studies primarily focus on the security of individual DL frameworks, but few of them can cope with the inconsistencies and security bugs during cross-framework conversion. Furthermore, the impact of these issues on DL applications remains largely unexplored. To this end, we propose TENSORSCOPE, a novel approach to test cross-framework APIs for security bugs. It takes as input a number of counterpart APIs that are supposed to be equivalent in functionality, then performs differential testing to identify the inconsistencies. We design novel strategies to boost testing efficiency, including 1) joint constraint analysis to raise the quality of test cases, and 2) error-guided test case fixing to refine the constraints for input. TENSORSCOPE is extensively evaluated on 1,658 APIs of six popular DL frameworks. The results show that TENSORSCOPE is more effective than FreeFuzz and DocTer by raising 28.7% and 24.3% code coverage, respectively. We find 257 bugs including 230 new bugs, and receive 8 CVEs and $1,100+ bounty with developers' acknowledgment. Most importantly, we make the first attempt to exploit these inconsistencies to make the accuracy of three models reduced by at most 3.5%.

1 Introduction

Deep learning is deployed in more and more real-world scenarios, such as image classification [43], face recognition [2] and autonomous vehicles [79]. Well known for its algorithmic weaknesses, researchers have extensively studied adversarial attack [65], model inversion attack [18], backdoor attack [10], etc. Recent studies [74, 83, 86] show that deep learning is also under threat of vulnerabilities in its underlying frameworks like TensorFlow and PyTorch. These vulnerabilities can either crash the execution of DL models, change the prediction results or even compromise the host machine of DL systems. Therefore, a secure framework is indispensable for the development and deployment of deep learning.

There is a surge of research on testing DL frameworks [14, 28, 57, 81]. DL frameworks receive neural networks as input and perform interpretation and optimization. Hence, studies [28, 80, 81] propose model-level testing approaches to evolve and mutate DL models, and then feed them to the frameworks, seeking for implementation bugs. However, it shows that model-level approaches [14, 82, 84] are limited in code coverage and can only find bugs triggered by neural networks. Therefore, API-level testing has drawn attention and significantly improved the efficiency of bug finding, where DL framework APIs are directly tested with specific cases.

It is worth noting that most of the prior works are targeting a single framework. However, the cross-platform deployment of DL models is gaining momentum since it can greatly widen the applicable scenarios. Many frameworks have their own converters for a handy switch with other frameworks, and open standards are also developed to ensure interoperability. In recent years, hundreds and thousands of issues related to cross-framework model conversion are raised in various DL framework repositories [24, 53]. Inconsistencies are more likely to arise when adapting a DL model to a new framework. Simply put, given one image, the classification model $\mathcal{M}$ can correctly recognize as l with confidence c. After being converted to model $\mathcal{M}'$ on another framework, it may only give a lower confidence c' (*i.e.*, $c' \leq c$) or even a wrong label l' (*i.e.*, $l' \neq l$). It motivates us to conduct research on identifying inconsistencies during cross-platform model deployment and explore how these inconsistencies affect model inference.

Challenges. In this study, we concentrate on testing cross-framework APIs. To achieve this goal, we have to solve the following challenges.

(1) *How to extract the constraints of API parameters and their implicit dependencies?* Obtaining accurate constraints

*Corresponding authors

for API parameters can benefit test case generation and improve testing efficiency. However, APIs may have semantic dependencies, serving as implicit constraints that cannot be extracted from API documents. Take the operator "tf.raw_ops .BatchMatMulV3(x,y,Tout,adj_x,adj_y,name)" in TensorFlow for example [76]. Besides that x and y are supposed to be of specific data types, the last dimension of x and the second last dimension of y should have the same size. This dependency is not described in the documentation, but without it, the majority of generated inputs are invalid.

(2) *How to generate representative test cases to find bugs more effectively?* It is non-trivial to generate high-quality test cases, especially corner cases, for a framework API, considering that the complex calling relations between APIs and the large value range of parameters. Traditional program analysis techniques like symbolic execution [7] and value range analysis [62] cannot be directly applied in this scenario. Moreover, code coverage is not effective in guiding the testing when a bug is only triggered by a specific value. That necessitates new indicators for measuring program states and providing meaningful feedback.

(3) *How to evaluate the hazards of the bugs and their exploitability in real-world scenarios?* Generally, operator APIs are well-encapsulated in DL frameworks and do not allow users to invoke directly. To exploit the bugs, attackers, in most cases, have to craft a specific model as input for DL frameworks. It brings a new intractable problem to determine whether the bug is reachable via this model. As a consequence, it is unclear what hazards the bugs can cause and their exploitability. To the best of our knowledge, there is no prior research on the systematic analysis of this problem.

Our Approach. To address the above challenges, we design a differential testing method for detecting inconsistency bugs of APIs across six DL frameworks. First, we prepare for inconsistency checking by extracting the counterpart APIs as test objects, by analyzing the equivalence conversion rules in each model converter (see Section 3.1). The conversion rules not only reveal the names of counterpart APIs in different frameworks, but also indicate the correspondence of API parameters. Then we extract the parameter constraints from both API profiles and implementations. From the API profiles, we are able to obtain constraints like supported types of each parameter and value ranges. From the code, we leverage static analysis to extract assertions and error-handling information that can help extract parameter dependencies mentioned in the first challenge. Moreover, the constraints are further refined if the program under test exits with an error message, for example, indicating a more accurate value range. Meanwhile, we collect all the constraints for counterpart APIs and make a joint analysis. To be specific, the intersection of constraints allows us to further narrow down the test scope and reach deeper code. The differences among constraints provide a higher chance to identify the value boundaries as well as corner cases that incur bugs (see Section 3.2). Last, we use three models in reality that are affected by buggy conversion and evaluate the hazards of the found bugs. Additionally, we implement a proof-of-concept attack, exploiting the bugs and quantifying their hazards (see Section 5.3).

We have evaluated TENSORSCOPE on 1,658 counterpart APIs extracted from six DL frameworks. The results show that TENSORSCOPE identifies 17,574 constraints, raising 24.7% compared to DocTer [84]. Meanwhile, TENSORSCOPE increases 54.5% code coverage compared to the well-known fuzzer Atheris [23], 28.7% to FreeFuzz [82], 24.3% to DocTer [84]. A total of 257 bugs are found, including 80 non-crash bugs and 177 crash bugs. All these bugs are recognized by developers, and 8 CVEs are obtained. The crash bugs are mainly distributed in segmentation fault (81.9%), floating point exception (7.9%), and abort (10.2%). We also conduct a manual inspection of these non-crash bugs and identify that the inconsistency bugs fall into three main categories: precision bugs (57.5%), data layout bugs (8.7%), and special value bugs (33.8%). In terms of bug hazard analysis, we demonstrate potential exploitation scenarios of these cross-framework bugs by model converters that decrease the accuracy of the three models by 2.3%-3.5%.

Contributions. We make the following contributions.

- Mapping. We review the conversion rules in the existing model converters and analyze the parameter mapping information of the counterpart API between frameworks. This information could be provided to framework developers for standardizing the design of framework APIs and providing users with more secure and effective interoperability when deploying models between different frameworks.
- Testing method. We propose a differential testing approach based on joint constraints and error message guidance, and implement a testing tool called TENSORSCOPE. This method combines different constraint information from multiple frameworks implementing the same functional API to generate test cases more efficiently. The approach helps testers find API implementation problems faster than with single framework testing.
- Testing results. We find 257 bugs, 230 are newly found and all of them are confirmed by the community and result in 8 CVEs. Furthermore, we perform a flawed model conversion on 3 models to show the possibility of doing attacks against inter-framework inconsistency problems with more practical and stealthy attack scenarios.

2 Background & Problem Statement

2.1 Deep Learning Framework

Deep learning frameworks like TensorFlow(Lite) [20] and PyTorch [52] have greatly propelled the development and deployment of DL applications. These frameworks provide handy

APIs for acquiring data, training models, serving predictions, and enhancing maintainability. Famous for their computation functions like data manipulation and automatic differentiation, developers can put more effort into the model design and training configuration. We categorize these APIs into two classes: *core APIs* that build the blocks of DL models. They are responsible for mathematical computations (*e.g.*, matrix multiplication, gradient calculation), model optimization (*e.g.*, parallelization) and cross-platform deployment (*e.g.*, cloud server, IoT devices); *interface APIs* that wrap core APIs and can be used directly by users. For example, TensorFlow provides multiple-language support including Python, Javascript, C++, and Java. These interface APIs usually invoke core APIs, *e.g.*, the tf.nn.conv2d Python API calls Conv2D C++ kernel through Python bindings.

Computational Graph. To ease the process of model training, DL frameworks propose the computational graph to describe how data flows and is computed. More formally, a computational graph $G = (N, E)$ is a directed acyclic graph, where the set of nodes N represents either operators or variables and E represents the flow of tensors between two nodes. Variables are a type of placeholder for persisting data that are often depicted as *tensors*. For simplicity, we refer to variables as tensors in this study. Tensors are multi-dimensional arrays and are depicted with dimensional information and data type (dtype). For example, <tf.Tensor:shape=(2,3),dtype=float64, ...> shows a tensor with a 2×3 matrix, its rank is 2 and its cell data is of float64. An operator is a basic unit for computation that takes as input a list of tensors and outputs the computation result. For example, tf.math.add shows an arithmetic calculation of element-wise addition that receives as input x and outputs y. Generally, the input needs to meet the requirements of "x.dtype==y.dtype" which are referred to as *constraints*.

2.2 Cross-framework Model Conversion

Recently, cross-framework model conversion gains its momentum due to the increasing need for interoperability [48]. It is the process of transforming a DL model from one framework format to another. On one hand, it is widely used in cross-framework deployment due to agile development and MLops [46]. Developers with TensorFlow experience can easily reuse PyTorch models with automatic converters. Considering this benefit of cross-framework conversion, many DL frameworks have implemented their own converters, supporting a smooth conversion to other popular frameworks. Additionally, ONNX [60] is developed with the purpose of model generalizability, which is an open standard for DL interoperability. It supports the conversion of many DL framework models into its own format, unifying the representation of models across different frameworks. Besides serving as the intermediate model, ONNX has its own inference engine named ONNXRuntime (ORT). Therefore, models in TensorFlow or PyTorch can be converted into ONNX and executed by ORT.

On the other hand, model conversion serves as an optimization for various deployment scenarios. For example, TensorFlow Lite is a lightweight framework for deploying DL models on mobile and edge devices. Models developed on TensorFlow are usually converted to semantic-equivalent models in the file format of TensorFlow Lite before deployment on edge devices. Compared to the format in TensorFlow (*i.e.*, protocol buffers), the model format in TensorFlow Lite (identified by flat buffers) is much more efficient in disk usage and model deserialization, especially for edge-device deployment.

New Threats. Bugs and vulnerabilities in DL frameworks have drawn researchers' attention in the past few years. According to the statistics of the CVE website, the number of vulnerabilities in TensorFlow is on the rise year by year [55]. In 2022, the TensorFlow framework has three times more memory corruption and bypass restriction vulnerabilities than the previous year. Therefore, uncovering and addressing these security issues is important to improve the security of deep learning applications. Even worse, the handy switch between different frameworks on top of model converters can further amplify the threat of these issues. It is non-trivial to implement a converter between two frameworks since 1) the quick evolution of DL frameworks makes converters more likely to be incompatible and acquires continuous efforts to follow the latest frameworks; 2) there may be no equivalent operators between two frameworks so that converters have to make an approximation to the target operator. These difficulties of DL framework and converter in implementation inevitably involve many bugs or even vulnerabilities, which are our main target in this study.

2.3 Problem Statement

Here we present an example to show that an equivalent conversion between different frameworks might induce problems, identified as inconsistent computation results.

```
logits = tf.random.uniform([1,10],
         dtype=tf.dtypes.half, maxval=100)
op = tf.raw_ops.LogSoftmax(logits=logits)
tf_model   = wrap_model(op)
tf_res     = tf_model.run()
onnx_model = tf2onnx(tf_model)

pt_model = onnx2pytorch(onnx_model)
pt_res   = torch_model.run()

np.testing.assert_allclose(tf_res, pt_res,
    rtol=1e-4, atol=1e-4)
# Mismatched elements: 9 / 10 (90%)
# Max absolute difference: 0.03884888
# Max relative difference: 0.00053648
```

Listing 1: Inconsistent results of equivalent conversion between TensorFlow and PyTorch

Motivation Example. As shown in Listing 1, the code first

builds a TensorFlow model containing a Logsoftmax operator (line 4). Then the model is converted into the ONNX format by converter *tf2onnx* (line 6), and further converted into PyTorch by converter *onnx2pytorch* (line 8). For a given random input parameter logits, the results of the TensorFlow model and the PyTorch model show a large difference that exceeds the reasonable error threshold 1×10^{-4} (line 11). The root cause is that when the *onnx2pytorch* converter does the model conversion, it converts the data type of the operator from float16 to float64. When large logits are encountered, the two sides produce inconsistent computation results. Since this operator is often used in the last layer of the model, its error can significantly affect the quality of inference results.

Moreover, we investigate the current inconsistency issues from a more comprehensive perspective. We search from multiple framework repositories the inconsistency issues with some keywords like "inconsistent", "inconsistency", "not consistent". We compile five reasons for how these inconsistencies occur as follows:

1. **Incorrect usage of APIs.** Developers may wrongly use specific APIs, leading to errors. For example, as shown in Listing 6, when passing the same parameters, the results of LRNGrad operators are inconsistent between CPU and GPU. That is, the gradient calculation results of the local response normalization (LRN) [44] are different. However, in reality, the developer claims that it is reasonable for CPU and GPU to return different gradients if the user passes invalid values to output_image and output_image must be the correct forward-pass output given input_image, *i.e.*, output_img should be the result of LRN processing of the input_img. If a user passes an invalid value for output_image, the API will return unexpected values.

2. **Incompatible versioning.** Many inconsistencies are caused by using obsoleted APIs [75]. As DL frameworks are evolving and iterating, their APIs are also inconstantly changed or even deprecated. The semantic differences across versions are eventually reflected in the differences in runtime results.

3. **Differences in dependency libraries**. This inconsistency is caused by numerical instability of dependent numerical libraries, *e.g.*, Eigen [27] in C++, Numpy [29] in Python. This inconsistency is more difficult to troubleshoot[1].

4. **Different Implementations.** Since an operator has different implementations and optimizations on different platforms and hardware, *e.g.*, "a*b/N" on CPU but "a/N*b" on GPU, they have different computational order, which leads to different loss of precision and finally inconsistent results[2]. This may not be an implementation error, but we cannot avoid this kind of corner case.

[1] https://github.com/tensorflow/tensorflow/issues/30995
[2] https://github.com/pytorch/pytorch/issues/87657

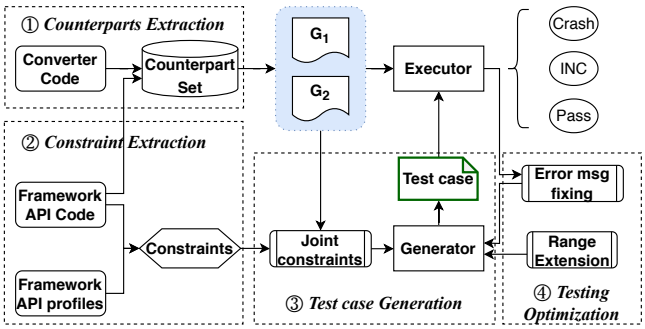

Figure 1: System Overview.

5. **Inconsistent model conversion.** To ease the switch among frameworks, each DL framework may offer handy APIs for automatic conversion. For example, PyTorch provides specific APIs for converting its model to ONNX, and ONNX is also equipped with conversion functions. These converters are found with logical errors, leading to inconsistencies between runtime frameworks.

This study focuses on the inconsistency of different DL frameworks in implementation. We aim to identify these issues through differential testing and then unveil what impact these issues can bring after model deployment. Here we only consider the last two types of inconsistencies since they stem from the incorrect implementation while the first three types are more related to developers' bad practices.

3 Approach

As depicted in Figure 1, we devise a differential testing approach with novel joint constraint analysis and error-guided testing optimization. It is divided into four steps: counterparts extraction, constraint extraction, test case generation, and optimization.

In counterparts extraction, we extract the cross-framework counterpart APIs by identifying conversion rules from model converters. One API may have its counterpart with a single API or a composition of multiple APIs in another framework. To facilitate comparison, TensorScope automatically wraps them into computational graphs, from which TensorScope can learn parameter correspondence for these counterpart APIs (see Section 3.1). In constraint extraction, we construct constraints for API parameters from both API profiles and implementation. (see Section 3.2). In test case generation, we perform a joint constraint analysis to refine the scope of API inputs. Based on that, we utilize an SMT solver to generate test cases for varying types of data (see Section 3.3). Finally, we propose two strategies to optimize our testing, *i.e.*, error-guided test case fixing and range extension to eliminate invalid cases dynamically captured during runtime and explore more input space to increase code coverage, respectively (see Section 3.4).

Table 1: Model converters between frameworks. The matrix is not symmetric, *e.g.*, converting from **TensorFlow** to **TensorFlow Lite** by *TOCO* is different from the backward converter *tflite2tf*. Additionally, some frameworks cannot be switched directly unless with an intermediate converter (denoted as *italics*), *e.g.*, converting PyTorch models to ONNX, then to TensorFlow models.

	TensorFlow (TF)	**TensorFlow Lite (TFL)**	**ORT**	**Mindspore (MS)**	**PyTorch**	**Paddle**
TensorFlow [19]	-	tflite2tf [37]	onnx2tf [36]	-	*onnx*	-
TensorFlow Lite [20]	TOCO [21]	-	*openvino*	-	*onnx*	-
ORT [54]	tf2onnx [58]	tf2onnx	-	-	torch.onnx.export [59]	-
Mindspore [34]	MindConverter [33]	-	MindConverter	-	MindConverter	-
PyTorch [52]	*onnx*	-	onnx2torch [17]	-	-	-
Paddle [5]	x2paddle [6]	-	x2paddle	-	x2paddle	-

3.1 Counterparts Extraction

Cross-framework model conversion assumes that frameworks share semantically equivalent functions. That is, one API can be replaced with one or more APIs in other frameworks. Here we call these equivalent APIs counterparts. In order to identify the inconsistencies of one model in different frameworks, we first determine convertible APIs and their counterparts.

Definition 1 *Given an API f, its counterpart is either one single API or a set of APIs, which can be loosely defined as $counterpart(f) = \{f_1, \ldots, f_n\}$, where f_i $(1 \leq i \leq n)$ is an API in another framework and $n \geq 1$.*

It has the following two properties:

- **Semantic equivalence.** With input x, the output of f and $\{f_1, \ldots, f_n\}$ should satisfy the following requirement:

$$\|f(x) - (f_1 \circ \ldots \circ f_n)(x)\| \leq \varepsilon \tag{1}$$

  where ε is a threshold to indicate the minimal distance between APIs and counterparts.

- **Sequentiality.** For a multi-API counterpart, the order of API combination matters, which should be explicitly regulated during conversion. For example, the counterpart of "AdjustContrastv2" in TensorFlow is {"Add", "Mul", "Sub", "ReduceMean"} in ONNX. Its function is to adjust the contrast of an image. The combination of these APIs should be "AdjustContrastv2(x0, x1) = Add(Mul(x1, Sub(x0, ReduceMean(x0))), ReduceMean(x0))".

Extracting counterparts across frameworks is challenging because the independence of each framework means that counterparts are likely to have different names and parameters. To this end, we propose to first identify candidate APIs as counterparts and then align parameters for validation.

```
registry: Dict[str, handler] =
{"onnx::AveragePool": PoolMapper,
 "onnx::MaxPool"    : PoolMapper, ...}
class PoolMapper(ONNXToMindSporeMapper):
  def _operation_name_in_ms(*args, **kwargs):
    if kwargs['op_name']=='onnx::AveragePool':
      op_name = 'nn.AvgPool{}d'
    else:
      op_name = 'nn.MaxPool{}d'
    dim = len(kwargs['params']['strides'])
    if dim == 3:
      return "P.MaxPool3D"
    return op_name.format(dim)
```

Listing 2: Conversion rules and handler of AveragePool and MaxPool operators in *mindconverter*

Candidate Identification. We first identify the candidate counterparts of one target API by parsing the equivalence conversion rules in each framework. An equivalence conversion rule is to transform one operator API in the source framework to the corresponding operator API(s) in the destination framework. From the 8 converters shown in Table 1, each converter has a mapping dictionary, also known as a registry, storing the above conversion rules (see Table 10 in Appendices B). The key of one registry is a source operator and the value is the handler for the conversion. As shown in Listing 2, the framework Mindspore implements its converter from ONNX in *mindconverter*. The operators *AveragePool* and *MaxPool* are processed with the handler PoolMapper. The handler makes conversions according to the name of operators (*i.e.*, op_name) and provided arguments (*i.e.*, dim). Usually, the call or name of the destination operator API usually appears in the handler code. So, we build a control flow graph (CFG) for each handler and carry out a lightweight context-aware static analysis. From one handler, we trace the execution paths in its CFG, recording the preconditions. More specifically, we traverse the CFG and assign the context for each branch, *i.e.*, in the context of what operator API. If one or more operators of the destination framework are reached, we label them as counterparts under certain preconditions. It is worth noting that this step requires human efforts for identifying the conversion handler for each converter, but can be easily adapted to other converters with the marked handler.

Different from DeepREL and EAGLE [80], we focus on cross-framework counterparts. To improve the test coverage of the framework code, we also collect the counterparts within a single framework by using DeepREL. However, our analysis reveals that many of the counterparts found by DeepREL are the target API's caller or callee. This phenomenon is very common in DL frameworks as APIs can be further extended with more functions or limited with shrunk input space by a wrapper. For example, the operator linalg.matmul in Tensor-

Flow is the caller of multiple primitive operators, including MatMul, BatchMatMul, and SparseMatMul. It does not make considerable contributions to finding real bugs, so we filter out these counterparts with direct invocation relationships.

Parameters Alignment. Counterpart APIs may have different numbers of parameters or the parameters are in different orders. Thus, we design an automated method to identify parameter correspondence between counterpart APIs for testing. We first wrap one side of a single API into a model, denoted as G_1. Then we convert G_1 into the other framework according to the model conversion rules to obtain G_2 (see Figure 2 for example). In this way, we build a pair of computational graphs that contains counterpart operator APIs. Generally, the model before and after the conversion does not change the nature of model inputs, including the shape, type, and order of the inputs. That is, the in- and out-degrees of the computational graph remain unchanged. Starting from the same input parameters in the two computational graphs, TensorScope follows the computation flow to locate the target API. It then observes the position of the parameters that the data is passed to, thus obtaining the corresponding relationship between the counterpart API parameters. We use a common parameter list to describe the common parameters in both counterpart APIs, $P_U = \{p_1, ..., p_l\}$ (l is the number of common parameters). For example, $P_U = \{p_1, p_2\}$ in Figure 2, p_1 corresponds to the dividend x in TruncateDiv and to the dividend *input* in torch.div, p_2 corresponds to the divisor y in the former and to the divisor *other* in the latter. For the parameters not in P_U, *e.g.*, the extra *rounding_mode* parameter in torch.div, we place them in the corresponding difference parameter list.

3.2 Constraint Extraction

To generate more valid test cases and improve code coverage, we extract two kinds of constraints for API parameters, *i.e.*, single-parameter (univariate) constraints C_s and multi-parameter (multivariate) constraints C_m from API profiles and implementations. Constraints C_p^F are defined on the parameters p of the API of framework F on five attributes, including type (*e.g.*, tensor, list, tuple), shape, data type (*e.g.*, int, half, float32), rank (specific for tensor type), and data value. Constraints are basically obtained from two sources as follows.

API Profiles. There are API profiles in DL frameworks, describing the types of parameters for each API. TensorScope automatically extracts the type constraints of APIs from their profiles (usually stored as *json* or *yaml* files). Taking the example in Figure 2, torch.div in PyTorch is the counterpart of TruncateDiv in TensorFlow. From the profiles, we can get the names and types of input parameters and outputs. As observed, the operator TruncateDiv has two input parameters x and y of type T. Type T can be many concrete values like DT_HALF and DT_FLOAT. So we finally extract $C_x.dtype \in T$, $C_y.dtype \in T$.

```
TORCH_META_FUNC(avg_pool3d) (
  ..., IntArrayRef kernel_size, ... ) {
  TORCH_CHECK(kernel_size.size() == 1 ||
    kernel_size.size() == 3,
    "avg_pool3d: kernel_size must be a single
    int, or a tuple of three ints"); ...}
```

Listing 3: Assertion of avg_pool3d operator in PyTorch

API Implementation. In addition to type constraints, we enrich the constraints by analyzing the statements of sanity checks in API implementation, *e.g.*, assertions. Assertions define constraints for API parameters, which hinder invalid test cases from reaching deep in code. To be specific, assertion statements (such as OP_REQUIRES and TORCH_CHECK) that validate inputs can be used as additional constraints. We summarize them in Table 11 in Appendices C. These statements have a specific syntax format within a DL framework. We record the position of the predicate expression in each assertion. For example, OP_REQUIRES is an assertion in TensorFlow, and its second parameter is a predicate expression, we parse this expression to an abstract syntax tree (AST). If the parsed items are the parameters of the current API and all other items are constants (*e.g.*, int), or Boolean function calls (*e.g.*, TensorShapeUtils::IsVector), we convert the expression as a constraint expression. In PyTorch, the first argument in the TORCH_CHECK macro is a predicate expression, and the second argument is an error message returned if the expression results false. TensorScope extracts the expression as new constraints for the current parameter, such as "kernel_size.size()==1||kernel_size.size()==3" in Listing 3, which is the constraint for the kernel_size parameter of the operator avg_pool3d. Besides, the constraints in the assertions are not limited to a single parameter, but also to multiple parameters, *e.g.*, the dtype of two parameters of TruncateDiv in Figure 2 must satisfy $C_x.dtype == C_y.dtype$ which means the dtype of both parameters x and y should be same.

Despite these measures, we may not be able to identify all the constraints for APIs. Therefore, we further supplement new ones from runtime errors (see Section 3.4).

3.3 Test Case Generation

After obtaining the constraint list $C = \{C_s, C_m\}$ for counterparts' parameters, TensorScope automatically generates random values for each parameter that satisfy their constraints. We formulate different generation rules according to the three types of parameters. ❶ Continuous. We generate random data using uniform sampling. ❷ Discrete. This type of data has a limited number of items. So we enumerate each item within its value domain. ❸ Categorical. This type of data has an unlimited number of items, *e.g.*, the names of operator APIs. In this case, we randomly generate data of a specific type with a fixed length (*e.g.*, 10).

Joint Constraints Analysis. The constraints of counterparts in different frameworks may be varying, which likely induce

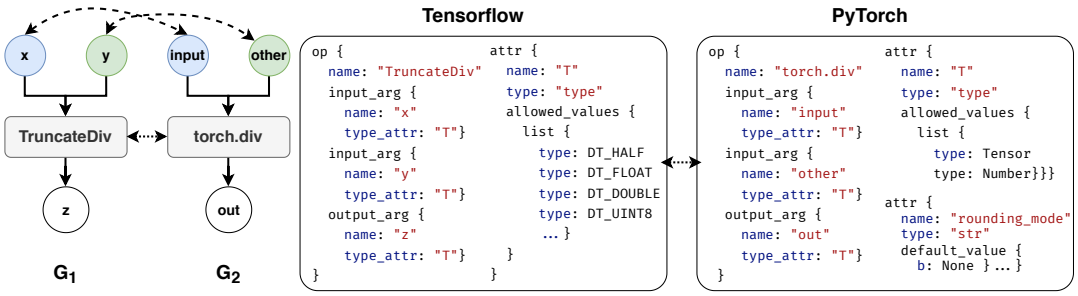

Figure 2: Parameter correspondence and constraints for counterpart APIs

inconsistencies. Therefore, we differentiate the constraints to generate more corner cases for testing. Given one shared parameter p_i in counterparts, we denote its constraints on the same parameter attribute as $C^A_{p_i}$ in framework A and $C^B_{p_i}$ in framework B. The intersection of $C^A_{p_i}$ and $C^B_{p_i}$ is computed as $C^U_{p_i}$, indicating that the value under these constraints (*i.e.*, $p_i \sim C^U_{p_i}$) are acceptable by both frameworks. As a consequence, the test cases under $C^U_{p_i}$ can reach deeper code. On the other hand, the differences in constraints between the two frameworks, *i.e.*, $C^A_{p_i} - C^B_{p_i}$ and $C^B_{p_i} - C^A_{p_i}$, suggest that the test cases in this space likely induce inconsistent results. For example, "at::cumsum(p1:input,p2:dim,...)" in PyTorch and "tf.cumsum(p1:x,p2:axis,...)" in TensorFlow are counterpart APIs. After extracting the constraints, we know that the parameter p_1 in at::cumsum should satisfy $C^A_{p_1}$ that its rank is in $(0,4]$ and $p_2 \in [-p_1.rank, p_1.rank)$. However, the $C^B_{p_1}$ in TensorFlow are that the parameter p_1's rank is in $(0,+\infty)$ and $p_2 \in [-p_1.rank, p_1.rank)$. The details are as follows:

$$C^A_{p_1} = \{p_1.rank \in (0,4] \wedge p_2 \in [-p_1.rank, p_1.rank)\}$$
$$C^B_{p_1} = \{p_1.rank > 0 \wedge p_2 \in [-p_1.rank, p_1.rank)\}$$
$$C^U_{p_1} = C^A_{p_1} \cap C^B_{p_1} = \{p_1.rank \in (0,4] \wedge p_2 \in [-p_1.rank, p_1.rank)\}$$
$$\Delta C^A_{p_1} = C^A_{p_1} - C^B_{p_1} = \phi$$
$$\Delta C^B_{p_1} = C^B_{p_1} - C^A_{p_1} = \{p_1.rank > 4\}$$

We begin by formulating the constraints on the parameters, p_i, of the counterpart APIs as two separate models. Then, we check the satisfiability of these models using an SMT solver. Subsequently, we determine the intersection and difference sets for these two constraint sets on the same attribute, and generate corresponding solutions (*i.e.*, test cases) for both the intersection and difference sets. To avoid duplication, we invert the current solution and incorporate it back into the constraint set, thereby generating a new test case.

3.4 Testing Optimization

During testing, we record the code coverage for each test case by runtime instrumentation for Python code and compile-time instrumentation for C++ code. Moreover, we construct

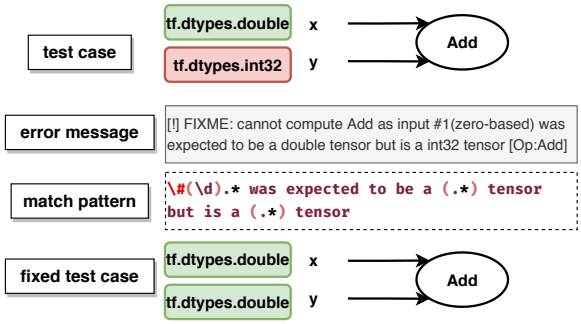

Figure 3: Test case fixing guided by error messages.

an oracle as follows to examine and sort testing results for counterpart APIs.

Test Oracle. If any of the APIs under test exit unexpectedly (*e.g.*, segmentation fault), we categorize this as *crash*. If any of them encounter an exception and throw error messages, we interpret this as an invalid test case, which is beneficial for test case optimization. Otherwise, if both counterparts execute successfully, we compare the results based on Equation 1. If the value distance exceeds ε, it is triaged as an *inconsistency*.

According to the runtime feedback, we propose two strategies to further optimize the testing.

```
if (handle->dtype != input_types[i]) {
  return errors::InvalidArgument(
    "cannot compute ", op->Name(), " as input
    #", i, "(zero-based)",
    " was expected to be a ", DataTypeString(
    input_types[i]),
    " tensor but is a ", DataTypeString(handle
    ->dtype), " tensor");}
```

Listing 4: Error handling code in TensorFlow

Error-guided Test Case Fixing. Figure 3 demonstrates that the program might throw error messages, indicating the invalidity of test cases. These messages can supplement parameter constraints and be used to refine the test cases. More generally, we first identify the throw point of an error message, which is typically an assertion or error handling statement. Then we analyze the message format and the preconditions associated with the error message. As shown in Figure 3, while

testing the Add operator, we encounter an error message that we trace back to the error handling code in Listing 4. The error message comprises a variable-length formatted string. We create a regular expression based on this format to extract key hints, such as the specific parameter "#1", the correct type input_types, and the current error type "handle->dtype". Also, we trace back from this error handling code to the first precondition: "handle->dtype!=input_types[i]". Inverting this condition gives us the correct condition, *i.e.*, changing the type to the expected correct type input_types.

Range Extension. This strategy seeks to increase code coverage by expanding the sampling space of numerical parameters to generate more test cases. After T tests, if the code coverage remains unchanged, we broaden the parameters' sampling range within the value space still under the constraints. This includes extending the value range and value shape. For instance, if the original value sampling range is $[min, max]$, it will be updated to $[min - mid, max + mid]$, where $mid = \frac{max-min}{2}$. The value shape will increase a dimension, *e.g.*, from (2,3) to (2,3,1). If the new range violates the parameter constraints, we halve mid until it satisfies the constraints. Meanwhile, the code coverage of the target API might have reached a high level, it is not the only metric we focus on during testing. There might still be hidden bugs, and it's infeasible to test all possible parameter value combinations, which is a significant challenge. Thus, we extend some special values like empty value, zero, negatives, minimum, and maximum for random selection and combination to accelerate bug finding.

Reconsidering from the common parameter list P_U, we note that aside from the parameters shared by both APIs, there are also differential parameters unique to each API. For example, the rounding_mode of torch.div in Figure 2. For these framework-unique parameters, we generate values according to their constraints to expand the testing space.

4 Evaluation

TensorScope is implemented in 3.2 KLOC of Python code which is accessible through [1]. We pull the latest version of the converter for analysis and testing (see Table 1 for links to each converter). The ε is usually taken as 1×10^{-4}. We assume that the API profiles and API implementation of a DL framework are consistent. We employ Semgrep [66] for Python and Weggli [22] for C++ to parse and extract constraints. Specifically, we use Z3Py [13] to solve the constraints extracted in Section 3.2. It is observed that the number of each parameter's constraints does not exceed 20, it is complexity-acceptable for Z3 solver. We compile each DL framework with the sanitizer feature enabled in the compilation options, including Address Sanitizer (ASan) [68] and Undefined Behavior Sanitizer (UBSan) [12]. This capability enables us to obtain a detailed explanation of the cause when a crash occurs. We also use the option (*e.g.*, -fcoverage-mapping) to collect code coverage information. For each API, we perform uniform sampling to generate a minimum of 20,000 test cases. During the range extension, the initial value of T is 1,000, which is doubled after each extension. We always run the buggy code five times and generate random values with the same seed to eliminate the disruptions caused by instability in the execution environment, avoiding flaky test cases.

Table 2: Statistics of counterpart pairs among six frameworks

	TF	TFL	ORT	MS	PyTorch	Paddle
TF	314	117	167	-	-	-
TFL	138	13	-	-	-	-
ORT	279	147	55	-	357	-
MS	-	-	60	241	85	-
PyTorch	-	-	97	-	281	-
Paddle	97	-	96	-	160	24

To evaluate the effectiveness and efficacy of TensorScope, we conduct extensive experiments and analysis, aiming to answer the following research questions.

RQ1. How accurate is the extraction of counterparts from different frameworks by TensorScope?

RQ2. How effective is TensorScope in extracting and conducting joint analysis of constraints?

RQ3. To what extent are the testing optimization strategies in TensorScope beneficial?

RQ4. What efficacy can TensorScope provide in bug finding?

RQ5. What advantages does TensorScope hold in comparison to related methods?

Experiment Subject and Settings. We select six mainstream DL frameworks with their latest versions shown in Table 7 and eight model converters (five official ones) shown in Table 1.

Runtime Environment. We use two Ubuntu 20.04 servers, one with Intel Xeon Gold CPU (64 cores) and the other with Intel Platinum CPU (192 cores) and two NVIDIA RTX 3090 GPUs) for the experiments.

4.1 Effectiveness in Counterparts Extraction

We extract a total of 12 groups of counterparts across six DL frameworks (refer to Table 2). For example, we extract 279 counterparts from TensorFlow to ORT, and 97 counterparts from ORT to PyTorch. Conversely, we extract 167 counterparts from ORT to TensorFlow, and 357 counterparts from PyTorch to ORT. The numbers vary as forward and backward converters are typically developed by different developers, each supporting a different number of convertible APIs. There are 2,336 one-to-one counterparts, 392 one-to-many

Table 3: Number of APIs in counterpart pairs among six frameworks

	TF	TFL	ORT	MS	PyTorch	Paddle	Total
API	515	147	175	289	365	167	1658
API_{total}	1028	159	186	655	498	236	2762
Ratio	50.1%	92.4%	**94.1%**	44.1%	73.3%	70.8%	**60.0%**

counterparts. The counterparts extracted from the converter code have no false positives or false negatives.

Additionally, we extract six groups of intra-framework counterpart APIs in gray cells to cover more non-popular APIs. After manual verification, we finally obtain 928 intra-framework counterpart APIs, excluding 46 false positives (yielding a false positive rate of 4.7%). Some APIs may share the same name, but their functionalities differ, or they may exist at different abstraction levels. For example, tf.keras.Conv2D and tf.nn.conv2d are equivalent in TensorFlow. But tf.keras.layers.Conv2D is not equivalent to tf.nn.conv2d, because the former is an interface API while the latter is a core API. The interface API casts the input tensor from float64 to float32 which will cause precision loss by default [78].

For an operator, it is not always directly convertible between two frameworks. It often requires the assistance of one or more intermediate frameworks. In total, there are 1,658 operator APIs from six frameworks in our counterparts database (see Table 3). The ratio of the counterpart APIs to the total number of APIs in each framework is as follows: TF (50.1%), TFL (92.4%), ORT (94.1%), MS (44.1%), PyTorch (73.3%), Paddle (70.8%). It is noteworthy that the ORT framework holds the highest proportion, suggesting superior interoperability. This indicates that models from other frameworks can be transformed into ORT with relative ease.

Table 4: Constraints extracted by TensorScope

	Single		Multiple		Total
	Python	C++	Python	C++	
TF	563	2943	1032	1948	6486
TFL	0	467	0	205	672
ORT	38	198	31	59	326
PyTorch	1186	2167	186	1332	4871
Paddle	207	1338	40	1660	3245
MS	499	892	229	354	1974

4.2 Effectiveness in Constraints Extraction & Joint Constraints Analysis

We successfully extract 17,574 constraints within 3 hours, which includes both C_s and C_m constraints shown in Table 4. Due to the guidance of error messages during dynamic testing (see Section 3.4), we verify and update the extracted constraints in a timely and accurate manner. Since there are multiple implementations of some operators, such as the same operator of ONNX exists in more than one opset [61], we take all the implemented constraints into account, resulting in some of them not being satisfied. We treat these unsatisfied, redundant constraints as false positives. The false positive rate at this stage is 7.3%. In total, we add 253 new constraints through the error message guidance, which we regard as false negatives. Because these constraints do not appear in the assertion statement, or they are located far from the operator function, a complex flow of data and control is passed between the operator implementation and the assertion statement. The false negative rate at this stage is 1.4%.

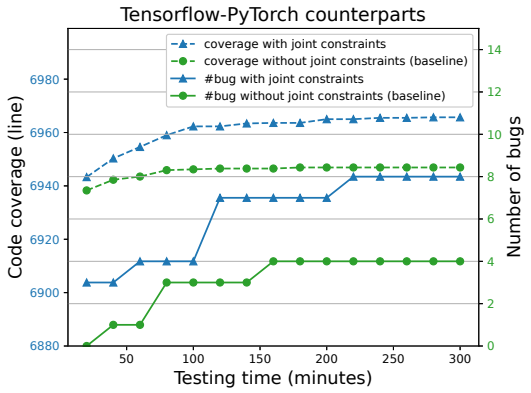

Figure 4: Comparison of code coverage and bug findings with and without joint constraint analysis among 20 TF-PT counterpart APIs.

Compared to DocTer on the same 1,409 APIs, TensorScope performs better in constraint extraction, identifying 11,357 constraints (8.1/API), as opposed to DocTer's 9,109 constraints (6.5/API). TensorScope finds 6,859 C_s and 4,498 C_m while DocTer only finds 6,382 C_s and 2,727 C_m. It shows that TensorScope has a distinct advantage in extracting C_m constraints. As for each type of constraints, TensorScope and DocTer extract 2,479/2,195 type constraints, 1,676/1,246 shape constraints, 3,788/3,555 data type constraints, 1,801/1,431 rank constraints, and 1,613/682 data value constraints, respectively. It shows that TensorScope achieves 12.9%, 34.5%, 6.6%, 25.9%, and 136.5% increases for five types of constraints. The remarkable gain for constraint *data value* stems from the fact that this type of constraint mostly resides in API implementation. However, DocTer only extracts constraints from API documents.

Here we compare the testing results of 20 counterparts with and without joint constraints, using the same testing configuration as in Section 4. We choose TensorFlow-PyTorch counterparts (converted through ONNX) as our targets and launch testing on the same set of randomly chosen 20 counterpart APIs. We compare both the number of found bugs and the code coverage of the counterpart APIs. From Figure 4 (X-axis represents the testing time, and Y-axis represents the line code coverage of counterpart APIs and the number of found bugs), we can see that the number of bugs found by

Table 5: Ablation study in testing optimization.

Settings	TensorFlow		PyTorch		Total	
	Cov.	#Bug	Cov.	#Bug	Cov.	#Bug
(i)	44872	12	42298	8	87170	20
(ii)	56019	12	54409	13	110428	25
(iii)	46072	20	45824	18	91896	38
(iv)	**58292**	**21**	**57297**	**22**	**115589**	**43**

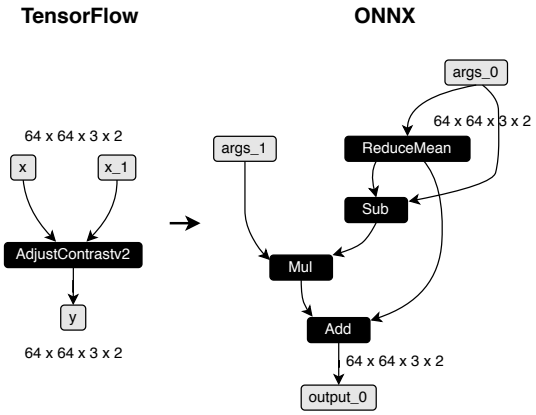

Figure 5: Conversion of the computational graph.

the tool with joint constraints is more than that without joint constraints, and the code coverage of counterpart APIs is also higher. Here the code coverage we adopt is the line coverage collected by coverage.py [8] and llvm-cov [49]. These results indicate that joint constraints can effectively guide the testing process and improve its effectiveness.

4.3 Effectiveness in Testing Optimization

In this section, we conduct an ablation study to evaluate the efficacy of two testing optimization strategies. We ask two authors to review all error messages (about 1 person-hour per framework), who design and cross-check 75 regular expressions that can recognize 98% error messages. These error messages help us find and refine 1,032 constraints. To evaluate the effects of these strategies, we conduct four experiments with distinct settings: (i) without optimization as the baseline; (ii) applying error-guided test case fixing exclusively; (iii) utilizing range extension in isolation; and (iv) integrating both strategies concurrently. We test TENSORSCOPE with different settings on 515 TensorFlow APIs and 365 PyTorch APIs for 6 hours. Table 5 shows that error messages can help get 23,258 more code coverage, and range extension can help find 18 more corner bugs. In total, these two testing optimization strategies help find 23 more bugs than the baseline.

Furthermore, we delve deeper into the specifics of range extension, which encompasses three distinct scopes, *i.e.*, value scope (17.5% coverage increase), shape scope (14.3%), and parameter scope (68.2%). It is observed that 3,000+ test cases generated with the extension in value scope allow us to find 11 bugs. It does not contribute much to code coverage but covers the boundary cases in the value domain space, including 5 extreme value bugs and 6 null bugs. Over 1,000 test cases generated in the shape scope allow us to test the scalability of the operator better, especially to find 2 more bugs related to high-dimensional tensors. Over 1,000 test cases generated in the parameter scope allow us to reach more branches and functionalities, yielding over 3,000 line coverage.

4.4 Efficacy

After 72-hours testing on these frameworks, we find a total of 257 bugs[3], including 80 non-crash bugs and 177 crash bugs. Among them, 230 bugs are newly found, and 188 are already resolved. We also obtain 8 CVEs, as detailed in Table 6. We verify and triage these bugs through manual efforts. The false positive rate at this stage is 1.9%, which can be divided into two types: ❶ uncaptured error messages, *e.g.*, not implement error, out-of-memory. ❷ stochastic algorithms, *e.g.*, we find that the random number generators in the two frameworks are different, leading to inconsistencies. There are 145 segmentation faults, 14 floating-point exceptions, and 18 aborts among these crash bugs as shown in Table 7. All non-crash bugs fall into the third category, which we further divide into three types. These non-crash bugs shown in Table 8 include 46 precision bugs, 7 data layout bugs, and 27 behavior bugs. We delve into a deeper analysis of these bugs in Section 5.

Case study. In TensorFlow, there is an operator named AdjustContrastv2 which adjusts the contrast of images. When the rank of the images parameter exceeds 3, the operator produces different results between TensorFlow and ORT (see Listing 5 in Appendices A). We measure the difference with absolute errors=11.99 and relative errors=0.18. We construct a TensorFlow model containing only operator AdjustContrastv2 and use *tf2onnx* for conversion. The conversion rules are shown in Listing 5, and the computational graphs before and after the conversion are shown in Figure 5. This is a one-to-many counterpart pair. The ONNX model uses four operators to implement the functions of the operator AdjustContrastv2, which are ReduceMean, Sub, Add, and Mul. This inconsistency arises because the converter erroneously reduces the axis of the images. It fails to account for the possibility of high-dimensional inputs in the images.

4.5 Comparison with Other Work

We compare our tool with three other works: Atheris [23], FreeFuzz [82], and DocTer [84]. Atheris is a well-known Python fuzzer. It is based on Libfuzzer [50] to test Python native code. We adopt the fuzzing code [77] based on Atheris developed by TensorFlow. Both FreeFuzz and DocTer are open-sourced. They mainly test TensorFlow and PyTorch

[3]Bug list is released at https://shorturl.at/cfimy

Table 6: Vulnerabilities found by TENSORSCOPE

ID	CVSS	Framework	Type	Symptom	Description
CVE-2022-35935	7.5	TensorFlow	missing validation	`CHECK` failure	given a **nonscalar** `num_results` value
CVE-2022-41883	7.5	TensorFlow	missing validation	OOB segfault	`indices` list shorter than the `data` list
CVE-2022-41899	7.5	TensorFlow	missing validation	segfault	given wrong shape tensors
CVE-2022-41891	7.5	TensorFlow	missing validation	segfault	element_shape=[]
CVE-2022-41897	7.5	TensorFlow	missing validation	Heap OOB	outsize inputs
CVE-2022-45907	9.8	PyTorch	code injection	arbitrary code execution	using dangerous `eval`
CVE-2022-45908	9.8	Paddle	code injection	arbitrary code execution	using dangerous `eval`
CVE-2022-46742	9.8	Paddle	code injection	arbitrary code execution	using dangerous `eval`

Table 7: Crash bugs found by TENSORSCOPE. "Segv" stands for segmentation fault, "FPE" is for floating point exception and "Abort" means program abort.

	Version	#Bug	Segv	FPE	Abort
TF	2.11	26	13	0	13
TFL	2.11	0	0	0	0
ORT	1.12.1	0	0	0	0
MS	1.9.0 & nightly	100	90	8	2
Paddle	develop	23	15	6	2
PyTorch	1.10.0 & 1.12.1	28	27	0	1
Total		**177**	**145**	**14**	**18**

Table 8: Inconsistency bugs found by TENSORSCOPE

	TF	TFL	ORT	MS	PyTorch	Paddle
TF	2	2	1	-	0	-
TFL	0	0	0	-	0	-
ORT	10	0	0	-	3	-
MS	0	0	5	3	3	-
PyTorch	24	0	14	-	1	-
Paddle	3	0	6	-	3	0

frameworks. Under the same time (5 hours) and environment, we randomly select 400 TensorFlow APIs and 400 PyTorch APIs and test them using these four tools. The results are shown in Table 9. All these found bugs are manually verified. It is observed that TENSORSCOPE detects the most number of bugs with 34 in TensorFlow and 30 in PyTorch. Also, TENSORSCOPE achieves high code coverage in both two frameworks. This success is due to the constraints collected from the code and the joint constraints used for testing. We also note that FreeFuzz identifies common program errors as bugs, which may produce false positives. For example, 3 *RuntimeError* bugs and 2 *NotImplementedError* bugs are thrown at Python runtime, making the program exit normally without triggering a crash. However, the identified bugs are crashes and numerical inconsistencies that are more harmful in practice (see Section 5).

Table 9: Comparison of code coverage and found bugs among Atheris, FreeFuzz, DocTer, and TENSORSCOPE.

	TensorFlow		PyTorch		Total	
	Cov.	#Bug	Cov.	#Bug	Cov.	#Bug
Atheris	32213	7	38502	4	70715	11
FreeFuzz	40702	13	44183	19	84885	32
DocTer	42877	18	45035	15	87912	33
TENSORSCOPE	**55362**	**24**	**53905**	**21**	**109267**	**45**

5 Measurement & Analysis

In this section, we present statistics for the found bugs and vulnerabilities, followed by bug types and hazard analysis.

5.1 Statistics of Bugs

The buggy APIs often occur in arithmetic modules, such as math (54), linalg (28). 56 of them are fused calculations with more than one primitive operator, primarily in the TensorFlow framework, *e.g.*, tf.raw_ops.CropAndResizeGradBoxes. These buggy APIs are typically used in various stages of DL model development, serving functions including data processing (98), arithmetic operations (58), gradient computation (15), loss functions (4), optimizers (2), and others (7). 32 operator APIs appear in commonly-used models, such as Logsoftmax in many classification models and so on. The root cause of most bugs is missing checks, accounting for 158 bugs, including checking data types (88), array boundaries (32), empty values (31) and others (7).

5.2 Types of Bugs

The bugs found by TENSORSCOPE can be categorized into two types. One can crash the code, and the other induces inconsistent computation results without any crash.

Inconsistent (non-crash) bugs. This type of bug does not cause a crash during testing. Therefore, they are only detected by comparing the computation results of these APIs in different frameworks. In this study, we obtain 80 in total for inconsistency bugs. After manual analysis, we categorize these bugs into three types: ❶ Precision bugs. We find 46 bugs of this type. Different frameworks may demand different computational precisions, in order to avoid the problem of numerical instability for complicated operators. For instance, TensorFlow often requires higher numerical precision (such as float64) for parameters. However, when converting a TensorFlow model to another framework like PyTorch, the converter may not meet these precision requirements (*e.g.*, it might use float32 for converted parameters), resulting in a decrease in precision. If the model contains multiple operators with precision bugs, the errors will accumulate and

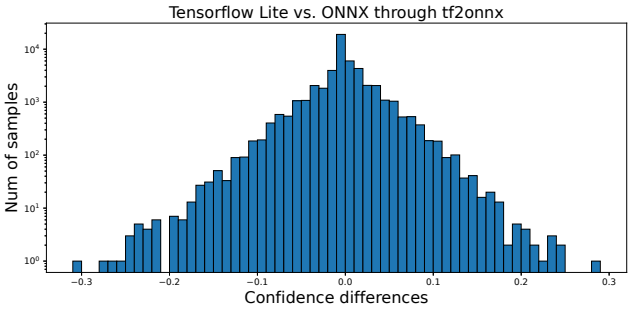

Figure 6: Distribution of confidence differences for the original classification result before and after conversion.

amplify during inference time, eventually leading to incorrect output. ❷ Data layout bugs. There are 7 bugs of this type. For instance, the tf.SpaceToBatchND operator requires the input data's dimension order to be NHWC [38]. However, the converter *x2Paddle* continues to use the NCHW data format, leading to the bug. ❸ Different exception handling bugs. There are 27 bugs in this type. Two counterpart APIs may behave differently about handling exceptions. One API may abort the program or return a unique error code for an exception. The returned error codes are varying significantly among frameworks, so they can also be captured by TensorScope. Different error codes make it impossible for users to make consistent judgments about the behavior of the model before and after conversion, and it becomes easy for users to miss processes that could harm subsequent programs. For example, when meeting invalid input not within the range $[-1, 1]$, tf.math.acos returns -2147483648 in TensorFlow, but torch.acos returns nan in PyTorch.

Bugs with crashes. We also find 177 bugs that can crash the program, some of which are identified as vulnerabilities (see Table 6). All of these bugs have been assigned CVE identifiers, given their potentially harmful impacts (*e.g.*, memory out-of-bounds (OOB) access, denial-of-service attacks, and other hazards). We also present their Common Vulnerability Scoring System (CVSS) scores, as calculated by NIST [56]. These vulnerabilities span a range of types, such as CWE-617 (Reachable Assertion), CWE-125 (OOB Read), CWE-20 (Improper Input Validation), CWE-682 (Incorrect Calculation), and CWE-77 (Command Injection). We present some examples of each vulnerability type in Appendices D.

5.3 Hazard Analysis

Unlike traditional software, vulnerabilities in DL frameworks can pose unique threats, especially degrading the security of DL applications. In this section, we propose a quantitative method for hazard analysis. For the first time, we successfully identify another type of adversarial example that "exploits" bugs in DL frameworks during the model conversion.

In the analysis, we select one of the flawed conversion rules found by TensorScope, which resides in the *tf2onnx* converter. This converter wrongly converts the data type of quantized operators from uint8 to float32. Many on-device models are quantized with uint8 data type [15]. If one model contains such quantized operators, the computation results will be inconsistent by 1×10^{-2} magnitude after conversion by this conversion rule. We select three models that can apply this conversion rule, of which the structures are MobileNet [32], InceptionV3 [71], and EfficientNet [73]. The classification accuracy is measured with the ImageNet validation set (50K images, 1K classes). The top-1 accuracy of MobileNet in TensorFlow is 72.3%, but after being converted to ONNX, this decreases to 68.8%. This conversion makes 1,741 samples from 134 classes misclassified (about 3.5% error rate). For example, an image of "snail" is correctly classified in the original model with confidence 0.7031. After the conversion, it is predicted as "bubble" with confidence 0.5938 by the model, and the confidence of class "snail" drops to 0.3984. This is a new type of adversarial example, caused by perturbations in the DL frameworks rather than in the input.

Figure 6 displays the distribution of changes in confidence for all correctly classified samples. As for the 17,113 samples with decreasing confidence, the reduction of confidence is 0.03 on average, with a maximum decrease of 0.30. To evaluate its transferability, we feed the 1,741 adversarial examples into the other two models (*i.e.*, InceptionV3 and EfficientNet), and 93.5% of them are misclassified.

The results demonstrate the possibility that the flaws in model converters can be exploited to make the converted model less robust during cross-framework development. More specifically, one attacker knows the bugs in a framework or a converter that can cause inconsistent results between the original model and its converted version. Then the attacker can craft a certain class of error samples, *i.e.*, adversarial samples specific to the target framework. These samples, while benign on the original framework, can cause damage once converted to another framework.

6 Discussion

Suggestions for developers. Here are three concrete suggestions for cross-framework developers. ❶ Beware of low-precision data types. Based on our findings, numerical computations with low-precision data can vary across different DL frameworks, platforms, and hardware. It is advisable to explicitly set the precision for input data and perform necessary precision checking. ❷ Use high-level APIs instead of low-level ones, as the former generally come with more guarantees. Framework developers should elaborate on the differences between these APIs, especially for cross-framework development. It can thus provide specific guidance when migrating higher-level APIs. ❸ Regularly update test cases for APIs, with a particular focus on rare and deprecated APIs. For uncommon operators, framework developers can provide

specific use cases and release test cases to guarantee their security. For deprecated APIs, framework developers should declare their status in the documentation and code in a timely manner to take mandatory update measures.

Limitations and future work. APIs that do not have counterparts cannot be tested with the joint constraint method but can be tested using their own constraints. The efficiency of test case generation via sampling is limited even within the constraints, which is also challenging. Additionally, in the hazard analysis, we only identify the examples that have different prediction results due to framework bugs. It is a type of untargeted attack. In the future, we aim to explore methods for passively perturbing the input and exploiting bugs for a targeted attack, such as manipulating an image so that it's classified as a specific label.

7 Related work

7.1 Deep Learning Framework Testing

In terms of test input, it can be roughly categorized into model-based testing and API-based testing.

Model-based testing. This line of approaches feeds data into various models to observe if any framework errors are triggered. CRADLE [64] is designed to identify and locate cross-framework inconsistencies among multiple backends. AUDEE [28] mainly focuses on evolving model parameters and input. It generates test cases based on the genetic algorithm and then finds inconsistencies among multiple DL frameworks. Luo *et al.* [51] propose new model coverage metrics using graph theory by varying DL models. Their test object is just a single framework. Furthermore, differential testing and metamorphic testing are also adapted for finding DL framework bugs. For instance, LEMON [81] leverages differential testing to boost the test of multiple back-end frameworks. Three kinds of model serialization mutation strategies are proposed for more diverse test inputs. Muffin [25] incorporates the model training phase into differential testing, designing appropriate metrics to measure discrepancies. EAGLE [80] identifies rules of equivalence for DL model mutation and uses metamorphic testing to identify bugs in individual frameworks. *Apart from the above studies, we select APIs as the test subject, which necessitates new test methods but enables us to test more framework functionalities.*

API-based testing. There is a growing body of research on fuzzing DL framework APIs. IvySyn [11] focuses on vulnerability discovery in APIs of individual frameworks, our work performs cross-framework testing. Some traditional API fuzzing approaches can be transferred to this new scenario. Pythia [3] is a fuzzing system for RESTful APIs that employs grammar rules and trains a *seq2seq* model to mutate test cases. A surge of research adapts API testing approaches to hunt DL framework bugs. DocTer [84] extracts API parameter constraints from the documents of DL frameworks with the help of natural language processing techniques to test APIs more accurately. DeepREL [14] considers API correlations within the same framework, mining and testing multiple groups of relational APIs based on single API signature similarity to validate their inconsistency. *Unlike the above studies,* TensorScope *proposes a joint API test across multiple frameworks, enabling us to construct more fine-grained constraints and identify inconsistencies more efficiently.*

7.2 Attacks and Defenses on DL Models

It is well known that DL models are suffering from multiple attacks such as adversarial attack [9, 31, 72], model inversion attack [70, 85], backdoor attack [4, 26, 47] and model extraction attack [30, 40]. These attacks motivate researchers to devise a series of methods to evaluate the security of DL models and further protect them. For example, Pei *et al.* propose DeepXplore [63] to find adversarial examples in DL models. Advdroid [15] analyzes the threats that exist in on-device DL models from two perspectives: physical theft and adversarial example attacks. In particular, the lack of protection in some frameworks leads models to be easily extracted and corrupted. Accordingly, a number of studies have emerged in the field of secure deep learning [41, 45, 67], such as Knott *et al.* [42] using secure multi-party computation to implement cryptographic inference. Recent attention is drawn to analyze the underlying DL frameworks such as [16, 35, 39, 69, 86]. These studies investigate and describe existing DL framework vulnerabilities, categorize the essential causes, and determine fixing solutions. However, most of the research is devoted to ensuring the quality of DL frameworks but is limited to security insights such as security influence and exploitability analysis of these framework vulnerabilities. *To the best of our knowledge, we are the first to conduct the exploitability analysis for these vulnerabilities in DL frameworks. It paves a new perspective for both security analysts and model developers to revisit vulnerabilities and the brought influence for DL models, which can further elevate the security of DL applications in critical areas.*

8 Conclusion

We propose a novel approach TensorScope to test 1,658 counterpart APIs across 6 DL frameworks and 8 model converters. Several optimization strategies are integrated including joint constraint analysis and error-guided test case fixing to enhance the quality of the test cases. TensorScope outperforms state-of-the-art approaches in terms of both code coverage and bug finding identified. Simply put, we have uncovered a total of 257 bugs, with 8 of them assigned CVE numbers. In addition, we analyze the root causes and potential hazards of these vulnerabilities in a real-world scenario.

Acknowledgments

We thank the shepherd and all the anonymous reviewers for their constructive feedback. The IIE CAS authors are supported in part by the National Key R&D Programmes of China (No. 2020AAA0104300), NSFC (92270204), Beijing Natural Science Foundation (No.M22004), Beijing Nova Program, Youth Innovation Promotion Association CAS, the Anhui Department of Science and Technology under Grant 202103a05020009 and a research grant from Huawei.

References

[1] TensorScope. https://github.com/tensorscopepro/Tensorscope, 2023.

[2] Insaf Adjabi, Abdeldjalil Ouahabi, Amir Benzaoui, and Abdelmalik Taleb-Ahmed. Past, present, and future of face recognition: A review. *Electronics*, 9(8):1188, 2020.

[3] Vaggelis Atlidakis, Roxana Geambasu, Patrice Godefroid, Marina Polishchuk, and Baishakhi Ray. Pythia: grammar-based fuzzing of rest apis with coverage-guided feedback and learning-based mutations. *arXiv preprint arXiv:2005.11498*, 2020.

[4] Eugene Bagdasaryan and Vitaly Shmatikov. Blind backdoors in deep learning models. In *30th USENIX Security Symposium (USENIX Security 2021)*, pages 1505–1521, 2021.

[5] Baidu. PaddlePaddle. https://www.paddlepaddle.org.cn/, 2022.

[6] Baidu. X2Paddle. https://github.com/PaddlePaddle/X2Paddle, 2022.

[7] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):1–39, 2018.

[8] Ned Batchelder. Coverage.py. https://github.com/nedbat/coveragepy, 2023.

[9] Nicholas Carlini and David A. Wagner. Towards evaluating the robustness of neural networks. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 39–57. IEEE Computer Society, 2017.

[10] Xinyun Chen, Chang Liu, Bo Li, Kimberly Lu, and Dawn Song. Targeted backdoor attacks on deep learning systems using data poisoning. *arXiv preprint arXiv:1712.05526*, 2017.

[11] Neophytos Christou, Di Jin, Vaggelis Atlidakis, Baishakhi Ray, and Vasileios P. Kemerlis. IvySyn: Automated Vulnerability Discovery in Deep Learning Frameworks. In *USENIX Security Symposium (USENIX Security 2023)*, 2023.

[12] Clang. UBSan. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html, 2022.

[13] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[14] Yinlin Deng, Chenyuan Yang, Anjiang Wei, and Lingming Zhang. Fuzzing Deep-Learning Libraries via Automated Relational API Inference. In *The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022.

[15] Zizhuang Deng, Kai Chen, Guozhu Meng, Xiaodong Zhang, Ke Xu, and Yao Cheng. Understanding Real-world Threats to Deep Learning Models in Android Apps. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 785–799, 2022.

[16] Kolegov Denis and Nikolaev Anton. Machine learning implementation security in the wild. *POC Conference*, 2019.

[17] Enot-ai. Onnx2torch. https://github.com/ENOT-AutoDL/onnx2torch, 2023.

[18] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. Model inversion attacks that exploit confidence information and basic countermeasures. In *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*, pages 1322–1333, 2015.

[19] Google. TensorFlow. https://www.tensorflow.org/, 2022.

[20] Google. TensorFlow Lite. https://www.tensorflow.org/lite, 2022.

[21] Google. TFLiteConverter. https://www.tensorflow.org/api_docs/python/tf/lite/TFLiteConverter, 2022.

[22] Google. Weggli. https://github.com/weggli-rs/weggli, 2022.

[23] Google. Atheris: A Coverage-Guided, Native Python Fuzzer. https://github.com/google/atheris, 2023.

[24] Google. Tensorflow conversion issues. https://github.com/search?q=repo%3Atensorflow%2Ftensorflow+conversion&type=issues, 2023.

[25] Jiazhen Gu, Xuchuan Luo, Yangfan Zhou, and Xin Wang. Muffin: Testing deep learning libraries via neural architecture fuzzing. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)*, pages 1418–1430, 2022.

[26] Tianyu Gu, Brendan Dolan-Gavitt, and Siddharth Garg. Badnets: Identifying vulnerabilities in the machine learning model supply chain. *CoRR*, abs/1708.06733, 2017.

[27] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[28] Qianyu Guo, Xiaofei Xie, Yi Li, Xiaoyu Zhang, Yang Liu, Xiaohong Li, and Chao Shen. Audee: Automated testing for deep learning frameworks. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 486–498. IEEE, 2020.

[29] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[30] Yingzhe He, Guozhu Meng, Kai Chen, Jinwen He, and Xingbo Hu. DRMI: A Dataset Reduction Technology based on Mutual Information for Black-box Attacks. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security 2021)*, 2021.

[31] Yingzhe He, Guozhu Meng, Kai Chen, Xingbo Hu, and Jinwen He. Towards security threats of deep learning systems: A sur-

vey. *IEEE Transactions on Software Engineering*, 48(5):1743–1770, 2020.

[32] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *CoRR*, abs/1704.04861, 2017.

[33] Huawei. Mindconverter. https://www.mindspore.cn/mindinsight/docs/en/master/migrate_3rd_scripts_mindconverter.html, 2022.

[34] Huawei. MindSpore. https://www.mindspore.cn/, 2022.

[35] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. Taxonomy of real faults in deep learning systems. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE)*, pages 1110–1121, 2020.

[36] Katsuya Hyodo. Onnx2tf. https://github.com/PINTO0309/onnx2tf, 2023.

[37] Katsuya Hyodo. Tflite2tensorflow. https://github.com/PINTO0309/tflite2tensorflow, 2023.

[38] Intel. Understanding Memory Formats. https://oneapi-src.github.io/oneDNN/dev_guide_understanding_memory_formats.html, 2022.

[39] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. A comprehensive study on deep learning bug characteristics. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 510–520, 2019.

[40] Mika Juuti, Sebastian Szyller, Samuel Marchal, and N. Asokan. PRADA: protecting against DNN model stealing attacks. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 512–527. IEEE, 2019.

[41] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. {GAZELLE}: A low latency framework for secure neural network inference. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1651–1669, 2018.

[42] Brian Knott, Shobha Venkataraman, Awni Hannun, Shubho Sengupta, Mark Ibrahim, and Laurens van der Maaten. Crypten: Secure multi-party computation meets machine learning. *Advances in Neural Information Processing Systems*, 34:4961–4973, 2021.

[43] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems (NIPS)*, pages 1106–1114, 2012.

[44] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, 2017.

[45] Nishant Kumar, Mayank Rathee, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. Cryptflow: Secure tensorflow inference. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 336–353. IEEE, 2020.

[46] Sam Leroux, Pieter Simoens, Meelis Lootus, Kartik Thakore, and Akshay Sharma. Tinymlops: Operational challenges for widespread edge ai adoption. In *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1003–1010. IEEE, 2022.

[47] Yuanchun Li, Jiayi Hua, Haoyu Wang, Chunyang Chen, and Yunxin Liu. Deeppayload: Black-box backdoor attack on deep learning models through neural payload injection. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 263–274. IEEE, 2021.

[48] Yu Liu, Cheng Chen, Ru Zhang, Tingting Qin, Xiang Ji, Haoxiang Lin, and Mao Yang. Enhancing the interoperability between deep learning frameworks by model conversion. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1320–1330, 2020.

[49] LLVM. llvm-cov. https://llvm.org/docs/CommandGuide/llvm-cov.html, 2022.

[50] LLVM. LibFuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html, 2023.

[51] Weisi Luo, Dong Chai, Xiaoyue Ruan, Jiang Wang, Chunrong Fang, and Zhenyu Chen. Graph-based fuzz testing for deep learning inference engines. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 288–299. IEEE, 2021.

[52] Meta. PyTorch. https://pytorch.org, 2022.

[53] Meta. PyTorch conversion issues. https://github.com/search?q=repo%3Apytorch%2Fpytorch+conversion&type=issues, 2023.

[54] Microsoft. ONNX Runtime. https://onnxruntime.ai/, 2022.

[55] MITRE. Google - Tensorflow - Vulnerability Statistics. https://www.cvedetails.com/product/53738/Google-Tensorflow.html?vendor_id=1224, 2022.

[56] NIST. Common Vulnerability Scoring System Calculator. https://nvd.nist.gov/vuln-metrics/cvss/v3-calculator, November 2022.

[57] Augustus Odena, Catherine Olsson, David Andersen, and Ian Goodfellow. Tensorfuzz: Debugging neural networks with coverage-guided fuzzing. In *International Conference on Machine Learning*, pages 4901–4911. PMLR, 2019.

[58] ONNX Official. Tf2onnx. https://github.com/onnx/tensorflow-onnx, 2022.

[59] PyTorch Official. torch2onnx. https://pytorch.org/docs/stable/onnx.html, 2022.

[60] ONNX. Open Neural Network Exchange. https://onnx.ai/, 2022.

[61] ONNX. The opset number of ONNX. http://onnx.ai/sklearn-onnx/auto_tutorial/plot_cbegin_opset.html, 2022.

[62] Jason RC Patterson. Accurate static branch prediction by value range propagation. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 67–78, 1995.

[63] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *proceedings of the 26th Symposium on Operating Systems Principles*, pages 1–18, 2017.

[64] Hung Viet Pham, Thibaud Lutellier, Weizhen Qi, and Lin Tan. Cradle: cross-backend validation to detect and localize bugs in deep learning libraries. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1027–1038. IEEE, 2019.

[65] Shilin Qiu, Qihe Liu, Shijie Zhou, and Chunjiang Wu. Re-

view of artificial intelligence adversarial attack and defense technologies. *Applied Sciences*, 9(5):909, 2019.

[66] R2c. Semgrep. https://github.com/returntocorp/semgrep, 2022.

[67] M Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M Songhori, Thomas Schneider, and Farinaz Koushanfar. Chameleon: A hybrid secure computation framework for machine learning applications. In *Proceedings of the 2018 on Asia conference on computer and communications security*, pages 707–721, 2018.

[68] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. {AddressSanitizer}: A fast address sanity checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 309–318, 2012.

[69] 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.

[70] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership inference attacks against machine learning models. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 3–18. IEEE Computer Society, 2017.

[71] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[72] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. Intriguing properties of neural networks. In *2nd International Conference on Learning Representations (ICLR)*, 2014.

[73] Mingxing Tan and Quoc V. Le. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. In *Proceedings of the 36th International Conference on Machine Learning (ICML)*, volume 97 of *Proceedings of Machine Learning Research*, pages 6105–6114. PMLR, 2019.

[74] Tensorflow Team. TensorFlow Security Advisories. https://github.com/tensorflow/tensorflow/tree/master/tensorflow/security, 2022.

[75] TensorFlow. Tensorflow issues-1. https://github.com/tensorflow/tensorflow/issues/47309, 2021.

[76] Tensorflow. The document of BatchMatMulV3 API. https://www.tensorflow.org/api_docs/python/tf/raw_ops/BatchMatMulV3, 2022.

[77] Tensorflow. TensorFlow Official Fuzzing. https://github.com/tensorflow/tensorflow/tree/master/tensorflow/security/fuzzing, 2023.

[78] TF_Support. Inequivalent output from tf.nn.conv2d and keras.layers.Conv2D. https://stackoverflow.com/questions/61087933/inequivalent-output-from-tf-nn-conv2d-and-keras-layers-conv2d, 2023.

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

[80] Jiannan Wang, Thibaud Lutellier, Shangshu Qian, Hung Viet Pham, and Lin Tan. Eagle: Creating equivalent graphs to test deep learning libraries. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pages 798–810, 2022.

[81] 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.

[82] Anjiang Wei, Yinlin Deng, Chenyuan Yang, and Lingming Zhang. Free lunch for testing: Fuzzing deep-learning libraries from open source. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pages 995–1007, 2022.

[83] Qixue Xiao, Kang Li, Deyue Zhang, and Weilin Xu. Security risks in deep learning implementations. In *2018 IEEE Security and Privacy Workshops (SPW)*, pages 123–128, 2018.

[84] Danning Xie, Yitong Li, Mijung Kim, Hung Viet Pham, Lin Tan, Xiangyu Zhang, and Michael W Godfrey. Docter: documentation-guided fuzzing for testing deep learning api functions. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 176–188, 2022.

[85] Minxing Zhang, Zhaochun Ren, Zihan Wang, Pengjie Ren, Zhumin Chen, Pengfei Hu, and Yang Zhang. Membership inference attacks against recommender systems. In *2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 864–879. ACM, 2021.

[86] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. An empirical study on tensorflow program bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 129–140, 2018.

Appendices

A Flawed Conversion Examples

We build a TensorFlow model using the code, "tf.raw_ops.AdjustContrastv2(images=tf.random.uniform([64,64,3,2],dtype=tf.dtypes.float32,maxval=255),contrast_factor=tf.random.uniform([],dtype=tf.dtypes.float32,maxval=1),)", then convert the model into an ONNX model by the following conversion handler of AdjustContrastv2 operator. The results shown in Line 21-22 between these two models are inconsistent.

```
1  # tf2onnx converter code:
2  @tf_op("AdjustContrastv2")
3  class AdjustContrastv2:
4    def version_1(cls, ctx, node, **kwargs):
5      images, contrast_factor = node.input
6      dtype = ctx.get_dtype(images)
7      if ctx.get_dtype(contrast_factor)!=dtype:
8        contrast_factor = ctx.make_node("Cast",
       [dtype], attr={'to': dtype}).output[0]
9      rank = ctx.get_rank(images)
10     utils.make_sure(rank is not None, "
       AdjustContrastv2 requires input of known
```

```
    rank")
    # Reduce everything except channels
    axes_to_reduce = list(range(rank))[:-1]
    mean = ctx.make_node("ReduceMean", [images
    ], attr={'axes': axes_to_reduce, 'keepdims
    ': True}, op_name_scope=node.name).output
    [0]
    diff = ctx.make_node("Sub", [images, mean
    ], op_name_scope=node.name).output[0]
    scaled = ctx.make_node("Mul", [diff,
    contrast_factor], op_name_scope=node.name)
    .output[0]
    result = ctx.make_node("Add", [scaled,
    mean], op_name_scope=node.name).output[0]
    ctx.replace_all_inputs(node.output[0],
    result)
    ctx.remove_node(node.name)

# Here is the inconsistent results:
# tf_res : [[[[154.93831  131.07579 ],
    [141.15346  162.48589 ], [123.68347
    143.64304 ]], ... ]]
# onnx_res : [[[[153.70927 , 126.60315 ],
    [139.92442 , 158.01324 ], [122.45444 ,
    139.1704  ]], ... ]]
```

Listing 5: Conversion of the AdjustContrastv2 operator between TensorFlow and ONNX

```
import tensorflow as tf
with tf.device('/GPU:0'):
    out = tf.raw_ops.LRNGrad(input_grads=a,
    input_image=x, output_image=y)
# [-0.29   0.97  -0.28]
with tf.device('/CPU:0'):
    out = tf.raw_ops.LRNGrad(input_grads=a,
    input_image=x, output_image=y)
# [2362.04 1360.11 2242.24]
```

Listing 6: Inconsistent results of LRNGrad operator between CPU and GPU

B Converter Registry & Handler

We manually identify registries and conversion handlers for 8 converters, as shown in Table 10. In this table, the wildcard represents all source/destination operators that can be converted.

Table 10: Registry & handler patterns in eight converters

Converter	Registry	Handler
tf2onnx	_OPS_MAPPING	tf_op
onnx2tf	ops.*	make_node
onnx2torch	_CONVERTER_REGISTRY	add_converter
MindConverter	*_to_ms.json	*Mapper
torch.onnx.export	symbolic_opset*	_onnx_symbolic
x2paddle	*map_ops	mapper
tflite2tf	op_new_types	make_graph
TOCO	OperatorType	Convert.*Operator

C Assertion Statements

We extract all types of assertions from configurations and manually identify 47 of them related to parameter constraints, covering 97% of all assertions. Table 11 presents these assertion patterns, where the wildcard represents a category of assertion macros. For instance, CHECKS* could represent CHECK, CHECK_EQ, among others. The "# Instances" represents the number of assertion instances utilized in each framework.

Table 11: Assertion patterns defined in six frameworks

Framework	Assertion patterns	# Instances
TF	OP_REQUIRES*	2
	CHECK*	5
	DCHECK*	5
TFL	TFLITE_CHECK*	4
	TFLITE_DCHECK*	4
ORT	ORT_RETURN_IF*	4
	ORT_ENFORCE	1
MS	MS_EXCEPTION*	4
PyTorch	TORCH_CHECK*	10
Paddle	PADDLE_ENFORCE*	8
Total		47

D Examples of Crash Bugs

Below, we provide a few representative examples for each vulnerability type.

- **Reachable Assertion.** In TensorFlow, there are many assertions and error-handling statements, *e.g.*, CHECK. The program will abort when it encounters an assertion constraint that is not satisfied. This could lead to Denial of Service (DoS) attacks. Some assertions (*e.g.*, DCHECK) are disabled in the release version. Consequently, encountering the same problematic input can trigger more severe issues. *E.g.*, CVE-2022-35935 in Table 6, the API forgets to check if the parameter num_results is a scalar which causes the program to exit unexpectedly.
- **OOB Read/Write.** OOB bugs in DL frameworks are often caused by tensor arrays that exceed their allocated memory during computation. *E.g.*, CVE-2022-41883 found between TensorFlow and PyTorch, when the size of indices does not match the size of data, causing a negative index to be calculated, resulting in OOB. These bugs could lead to DoS attacks, or even more severe harm, *e.g.*, Heap OOB in CVE-2022-41897.
- **Floating Point Exception.** There are 28 bugs that can cause floating point exception, broken down as follows: invalid operation (10.7%), division by zero (42.8%), overflow (21.5%), and underflow (25.0%). In fact, DL frameworks sometimes lack corner-case checking, resulting in floating point exceptions and DoS attacks. Many arithmetic APIs in Paddle, such as paddle.linalg.svd, paddle.linalg.eig and pad

Table 12: Reporting and reward methods by DL frameworks. The **digits** in the parenthesis are the number of bugs reported.

	Reporting			Reward		
	E-mail (129)	Form (4)	Issue (8)	CVE	Bounty	SA
TensorFlow	✗	✔	✔	✔	✔	✔
PyTorch	✗	✔	✔	-	✔	✗
Paddle	✔	✗	✔	✔	✗	✔
Mindspore	✔	✗	✔	✔	✔	✔

dle.linalg.lu, do not check whether a parameter is empty, but their counterpart APIs in PyTorch do include these checks. Using our differential method, we quickly identify these missing checks. When we pass empty tensors generated by "paddle.to_tensor(np.random.uniform(-10,10,[0,0]).astype(np.float64))" to them, they throw a floating point exception and exit uncleanly.

- **Command Injection.** Command injection can also occur in DL frameworks due to the misuse of dangerous functions. Some functions (*e.g.*, functions that are used to parse input strings or files) require the execution of a string during development. As a result, developers haphazardly use the function eval in Python to achieve this purpose. But eval is dangerous especially when its parameter is exposed to users without checking. Based on this, we find three critical vulnerabilities, CVE-2022-45907, CVE-2022-45908, and CVE-2022-46742 in PyTorch and Paddle. All of them lead to arbitrary code execution due to using eval blindly.

E Response from DL frameworks.

We have reported the bugs to the maintainers of four DL frameworks, including TensorFlow, PyTorch, Paddle, and Mindspore. Through multiple rounds of interaction, we unveil some unknown measures of the maintainers managing these bugs.

First, there are mainly three ways of reporting bugs: *e-mail*, *web form*, and *github/gitee issue*. Table 12 shows what reporting methods are supported by DL frameworks. Second, DL frameworks do not reach a consensus on the severity of bugs. For example, we reported two bugs of code injection to PyTorch and Paddle, respectively. One of them did not take the report seriously, considering the difficulty of triggering the bug, while the other patched it quickly and assigned a new CVE. Additionally, different DL frameworks have varying reward/acknowledgment methods towards bugs, such as issued CVEs, paid bounty, and acknowledgment in their security advisories. Finally, the accountability of bugs varies from bug type. Generally, crash bugs are acknowledged by framework maintainers, but it is somehow unclear to assign which maintainers for inconsistency bugs. Considering the incomplete equivalence between DL frameworks, converters have to bridge the gap through approximation methods. Therefore, based on our observations, most framework maintainers attribute inconsistency bugs to converters.