Reliability Assurance for Deep Neural Network Architectures Against Numerical Defects

Linyi Li* Yuhao Zhang† Luyao Ren‡§ Yingfei Xiong‡§ Tao Xie‡§

*Department of Computer Science, University of Illinois Urbana-Champaign, *linyi2@illinois.edu*
†Department of Computer Sciences, University of Wisconsin-Madison, *yuhao.zhang@wisc.edu*
‡School of Computer Science, Peking University, *rly@pku.edu.cn*, *xiongyf@pku.edu.cn*, *taoxie@pku.edu.cn*
§Key Laboratory of High Confidence Software Technologies, Ministry of Education (Peking University)

***Abstract*—With the widespread deployment of deep neural networks (DNNs), ensuring the reliability of DNN-based systems is of great importance. Serious reliability issues such as system failures can be caused by numerical defects, one of the most frequent defects in DNNs. To assure high reliability against numerical defects, in this paper, we propose the RANUM approach including novel techniques for three reliability assurance tasks: detection of potential numerical defects, confirmation of potential-defect feasibility, and suggestion of defect fixes. To the best of our knowledge, RANUM is the first approach that confirms potential-defect feasibility with failure-exhibiting tests and suggests fixes automatically. Extensive experiments on the benchmarks of 63 real-world DNN architectures show that RANUM outperforms state-of-the-art approaches across the three reliability assurance tasks. In addition, when the RANUM-generated fixes are compared with developers' fixes on open-source projects, in 37 out of 40 cases, RANUM-generated fixes are equivalent to or even better than human fixes.**

***Index Terms*—neural network, numerical defect, testing, fix**

I. Introduction

Deep Neural Networks (DNNs) are successfully deployed and show remarkable performance in many challenging applications, including facial recognition [20, 54], game playing [37, 44], and code completion [19, 4]. To develop and deploy DNNs, one needs to attain a DNN architecture, which is usually encoded by program code as the example shown in Figure 2. First, for training, the user executes the program with the architecture on the given training/validation data, attains the model weights, and stores them in a weight file. The architecture along with the weights is named a model. Then, for inference, the user loads the weight file to CPU/GPU memory or AI chips, executes the same program with the given inference sample and weights as arguments, and gets the model prediction result as program output. With the wide deployment of DNN models (resulted from training DNN architectures), reliability issues of DNN-based systems have become a serious concern, where malfunctioning DNN-based systems have led to serious consequences such as fatal traffic accidents [23].

To assure reliability of DNN-based systems, it is highly critical to detect and fix numerical defects for two main reasons. First, numerical defects widely exist in DNN-based systems. For example, in the DeepStability database [14], over 250 defects are identified in deep learning (DL) algorithms where over 60% of them are numerical defects. Moreover, since numerical defects exist at the architecture level, any model using the architecture naturally inherits these defects. Second, numerical defects can result in serious consequences. Once numerical defects (such as divide-by-zero) are triggered, the faulty DNN model will output NaN or INF instead of producing any meaningful prediction, resulting in numerical failures and system crashes [58, 56]. Thus, numerical defects hinder the application of DNNs in scenarios with high reliability and availability requirements such as threat monitoring in cybersecurity [32] and cloud systems controlling [35, 13].

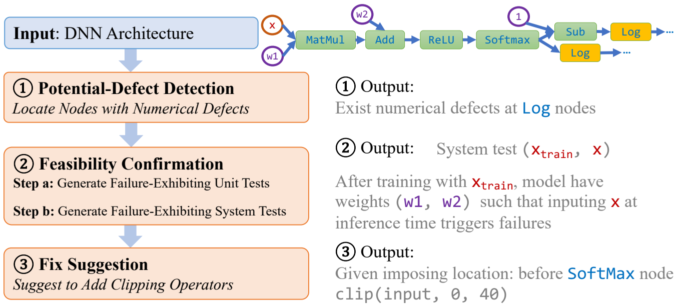
Input: DNN Architecture
x
w1
w2
MatMul
Add
ReLU
Softmax
1
Sub
Log
Log
① Potential-Defect Detection
Locate Nodes with Numerical Defects
② Feasibility Confirmation
Step a: Generate Failure-Exhibiting Unit Tests
Step b: Generate Failure-Exhibiting System Tests
③ Fix Suggestion
Suggest to Add Clipping Operators
① Output:
Exist numerical defects at Log nodes
② Output: System test (x_train, x)
After training with x_train, model have weights (w1, w2) such that inputing x at inference time triggers failures
③ Output:
Given imposing location: before SoftMax node
clip(input, 0, 40)

Fig. 1. Workflow for reliability assurance against numerical defects in DNN architectures. The left-hand side shows three tasks and the right-hand side shows corresponding examples. RANUM supports all three tasks, and is the first automatic approach for system test generation and fix suggestion.

To address numerical defects in DNN architectures, in this paper, we propose a workflow of reliability assurance (as shown in Figure 1), consisting of three tasks: potential-defect detection, feasibility confirmation, and fix suggestion, along with our proposed approach to support all the three tasks.

Potential-Defect Detection. In this task, we detect all potential numerical defects in a DNN architecture, with focus on inference-phase numerical defects (which are operators that potentially exhibit numerical failures in the inference phase) for two main reasons, following the literature [59, 52]. First, inference-phase numerical defects can be triggered after the model is deployed and thus are more devastating than training-phase defects [26, 59]. Second, training-phase numerical defects can usually be detected during the inference phase. For example, the type of training-phase NaN gradient defect is caused by an operator's input that leads to invalid derivatives, where this input also triggers failure the inference phase [52].

Feasibility Confirmation. In this task, we confirm the feasibility of these potential numerical defects via generating failure-exhibiting system tests. As shown in Figure 1, a system test is a tuple of training example[1] $\boldsymbol{x}_{\mathsf{train}}$ and inference example $\boldsymbol{x}$ such that after the training example is used to train the architecture under consideration, applying the resulting model on the inference example exhibits a numerical failure.

Fix Suggestion. In this task, we fix a feasible numerical defect. To determine the fix form, we have inspected the developers' fixes of the numerical defects collected by Zhang et al. [58] by looking at follow-up Stack Overflow posts or GitHub commits. Among the 13 numerical defects whose fixes can be located, 12 fixes can be viewed as explicitly or implicitly imposing interval preconditions on different locations, such as after inputs or weights are loaded and before suspicious operators (ones that potential defects are associated with) are invoked. Thus, imposing an interval precondition, e.g., by clipping (i.e., chopping off the input parts that exceed the specified input range) the input for suspicious operator(s), is an effective and common strategy for fixing a numerical defect. Given a location (i.e., one related to an operator, input, or weight where users prefer to impose a fix), we suggest a fix for the numerical defect under consideration.

To support all three tasks of the **r**ealiability **a**ssurance process against DNN **num**erical defects, we propose the **RANUM** approach in this paper.

For task ① and task ②a, which are already supported by two existing tools (DEBAR [59] and GRIST tool [52]), RANUM introduces novel extensions and optimizations that substantially improve the effectiveness and efficiency. (1) The state-of-the-art tool for potential-defect detection is DEBAR. However, DEBAR can handle only static computational graphs and does not support widely used dynamic graphs in PyTorch programs [27]. RANUM supports dynamic graphs thanks to our novel technique of *backward fine-grained node labelling*. (2) The recent GRIST tool generates failure-exhibiting unit tests for confirming potential-defect feasibility by gradient back-propagation, but uses the original inference input and weights as the starting point. However, recent studies [22, 9] on DNN adversarial attacks suggest that using a randomized input as the starting point leads to stronger attacks than using the original input. Taking this observation, we combine gradient back-propagation with random initialization in RANUM.

For task ② and task ③, which are not supported by any existing tool, RANUM is the first automatic approach for them.

For **feasibility confirmation**, RANUM is the **first** approach that generates failure-exhibiting **system** tests that contain training examples. Doing so is a major step further from the existing GRIST tool, which generates failure-exhibiting unit tests ignoring the practicality of generated model weights. Given that in practice model weights are determined by training examples, we propose the technique of *two-step generation* for this task. First, we generate failure-exhibiting unit tests. Second, we generate a training example that leads to the model weights in the unit test when used for training. For the second step, we transform the task into finding an input that leads to specific model gradients, where we extend the deep-leakage-from-gradient (DLG) attack [61] by incorporating the straight-through gradient estimator [3].

For **fix suggestion**, RANUM is the **first** automatic approach. RANUM is based on the novel technique of *abstraction optimization*. We observe that a defect fix in practice is typically imposing interval clipping on some operators such that each later-executed operator (including those suspicious ones) can never exhibit numerical failures. Therefore, we propose the novel technique of abstraction optimization to "deviate away" the input range of a suspicious operator from the invalid range, falling in which can cause numerical failures.

For RANUM, we implement a tool (all artifacts including source code are attached to this submission) and evaluate it on the benchmarks [52] of 63 real-world DNN architectures containing 79 true numerical defects, which are the largest benchmarks of DNN numerical defects to the best of our knowledge. The evaluation results show that RANUM is both effective and efficient in all three tasks for DNN reliability assurance. (1) For potential-defect detection, RANUM detects >60% more true defects than the state-of-the-art DEBAR approach. (2) For feasibility confirmation, RANUM generates failure-exhibiting unit tests to confirm potential numerical defects in the benchmarks with 100% success rate, whereas with the much higher time cost (17.32X), the state-of-the-art GRIST approach generates unit tests to confirm defects with 96.96% success rate. More importantly, for the first time, RANUM generates failure-exhibiting system tests that confirm defects (with 92.78% success rate). (3) For fix suggestion, RANUM proposes fix suggestions for numerical defects with 100% success rate. In addition, when the RANUM-generated fixes are compared with developers' fixes on open-source projects, in 37 out of 40 cases, RANUM-generated fixes are equivalent to or even better than human fixes.

This paper makes the following main contributions:

- We formulate the reliability assurance problem for DNN architectures against numerical defects and elaborate three important tasks for this problem.
- We propose RANUM—the first automatic approach that solves all these three tasks. RANUM includes three novel techniques (backward fine-grained node labelling, two-step test generation, and abstraction optimization) and solves system test generation and fix suggestion for the first time.
- We implement RANUM and apply it on 63 real-world DNN architectures, showing the high effectiveness and efficiency of RANUM compared to both the state-of-the-art approaches and developers' fixes.

II. Background and Approach Overview

In this section, we introduce the background of DNN numerical defects and failures, and then give an overview of the RANUM approach with a running example.

A. Background

DL developers define DNN architecture with code using modern DL libraries such as PyTorch [27] and Tensorflow [1].

[1] In real settings, multiple training examples are used to train an architecture, but generating a single training example to exhibit failures (targeted by our work) is desirable for ease of debugging while being more challenging than generating multiple training examples to exhibit failures.

```
input_data = tf.placeholder("float", [1, n_features], name='x-input')
input_labels = tf.placeholder("float", [1, n_classes], name='y-input’)
self.W_ = tf.Variable(tf.zeros([n_features, n_classes]),
                      name='weights')
self.b_ = tf.Variable(tf.zeros([n_classes]),
                      name='biases')
model_output = tf.nn.softmax(tf.matmul(input_data, self.W_) +
                             self.b_)
cost = -tf.reduce_mean(input_labels * tf.log(model_output) +
                       (1 - input_labels) * tf.log(1 - model_output),
                       name='cost')
self.obj_function = tf.reduce_min(tf.abs(model_output),
                                  name='obj_function')
```

Fig. 2. A DL program snippet that defines a linear regression model from real-world numerical defect benchmarks (Case 2a in [52]).

The DNN architecture can be expressed by a computational graph. Figures 2 and 3 depict a real-world example. Specifically, the DNN architectures in DL programs can be automatically converted to ONNX-format computational graph [6].

The computational graph can be viewed as a Directed Acyclic Graph (DAG): $\mathcal{G} = \langle \mathcal{V}, \mathcal{E} \rangle$, where $\mathcal{V}$ and $\mathcal{E}$ are sets of nodes and edges respectively. We call nodes with zero in-degree as *initial nodes*, which correspond to input, weight, or constant nodes. Initial nodes provide concrete data for the DNN model. The data from each node is formatted as a tensor, i.e., a multidimensional array, with a specified data type and array shape annotated alongside the node definition. We call nodes with positive in-degree as *internal nodes*, which correspond to concrete operators, such as matrix multiplication (`MatMul`) and addition (`Add`). During model training, the model weights, i.e., data from weight nodes, are generated by the training algorithm. Then, in deployment phase (i.e., model inference), with these trained weights and a user-specified input called inference example, the output of each operator is computed in topological order. The output of some specific node is used as the prediction result.

We let $\boldsymbol{x}$ and $\boldsymbol{w}$ to denote the concatenation of data from all input nodes and data from all weight nodes respectively.[2] For example, in Figure 3, $\boldsymbol{x}$ concatenates data from nodes 1 and 11; and $\boldsymbol{w}$ concatenates data from nodes 2 and 4. Given specific $\boldsymbol{x}$ and $\boldsymbol{w}$, the input and output for each node are deterministic.[3] We use $f_n^{\text{in}}(\boldsymbol{x};\boldsymbol{w})$ and $f_n^{\text{out}}(\boldsymbol{x};\boldsymbol{w})$ to express input and output data of node n, respectively, given $\boldsymbol{x}$ and $\boldsymbol{w}$.

Numerical Defects in DNN Architecture. We focus on inference-phase numerical defects. These defects lead to numerical failures when specific operators receive inputs within invalid ranges so that the operators output `NaN` or `INF`.

Definition 1. For a given computational graph $\mathcal{G} = \langle \mathcal{V}, \mathcal{E} \rangle$, if there is a node $n_0 \in \mathcal{V}$, such that there exists a valid input and valid weights that can let the input of node n_0 fall within invalid range, we say there is a ***numerical defect*** at node n_0.

$$\text{Formally}, \exists \boldsymbol{x}_0 \in \mathcal{X}_{\mathsf{valid}}, \boldsymbol{w}_0 \in \mathcal{W}_{\mathsf{valid}}, f_{n_0}^{\text{in}}(\boldsymbol{x}_0;\boldsymbol{w}_0) \in \mathcal{I}_{n_0,\mathsf{invalid}}$$
$$\Longrightarrow \exists \text{ numerical defect at node } n_0.$$

In the definition, $\mathcal{X}_{\mathsf{valid}}$ and $\mathcal{W}_{\mathsf{valid}}$ are valid input range and weight range respectively, which are clear given the deployed scenario. For example, ImageNet `Resnet50` model has valid

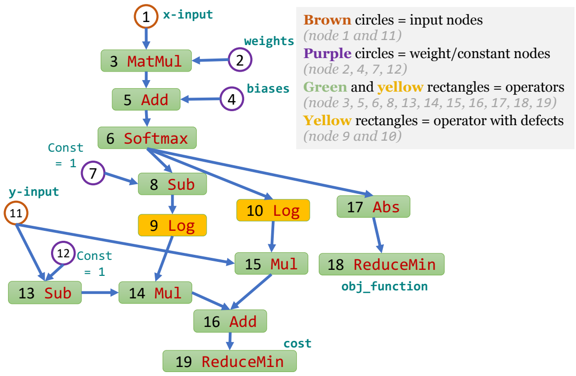

Fig. 3. Computational graph encoded by Figure 2 snippet.

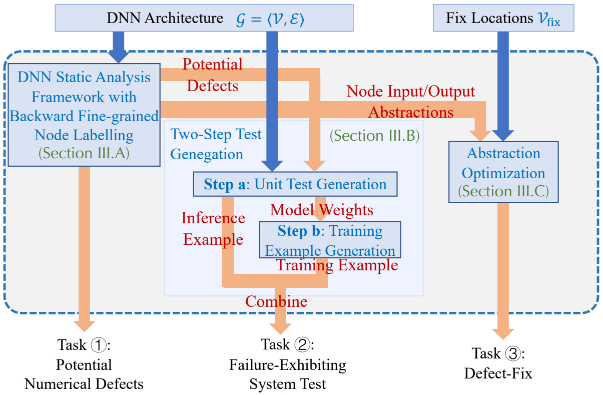
DNN Architecture $\mathcal{G} = \langle \mathcal{V}, \mathcal{E} \rangle$
Fix Locations $\mathcal{V}_{\text{fix}}$
DNN Static Analysis Framework with Backward Fine-grained Node Labelling (Section III.A)
Potential Defects
Node Input/Output Abstractions
Two-Step Test Genegation
(Section III.B)
Step a: Unit Test Generation
Inference Example
Model Weights
Step b: Training Example Generation
Training Example
Combine
Abstraction Optimization (Section III.C)
Task ①: Potential Numerical Defects
Task ②: Failure-Exhibiting System Test
Task ③: Defect-Fix

Fig. 4. Overview of the RANUM approach.

input range $\mathcal{X}_{\mathsf{valid}} = [0,1]^{3\times224\times224}$ since image pixel intensities are within $[0,1]$, and valid weight range $\mathcal{W}_{\mathsf{valid}} = [-1,1]^p$ where p is the number of parameters since weights of well-trained `Resnet50` models are typically within $[-1,1]$. The invalid range $\mathcal{I}_{n_0,\mathsf{invalid}}$ is determined by n_0's operator type with detailed definitions in Suppl. B. For example, for `Log` operator, the invalid range $\mathcal{I}_{n_0,\mathsf{invalid}} = (-\infty, U_{\min})$ where $U_{\min}$ is the smallest positive number of tensor's data type.

B. Approach Overview

In Figure 4, we show the overview structure of the RANUM approach. RANUM takes a DNN architecture as the input. First, the DNN static analysis framework in RANUM detects all potential numerical defects in the architecture, which is the task ① in Figure 1. Second, the two-step test generation component (gradient back-propagation + extended DLG attack) confirms the feasibility of these potential numerical defects, which is the task ② in Figure 1. Third, the abstraction optimization component takes the input/output abstractions produced by the DNN static analysis framework along with the user-specified fix locations and produces preconditions to fix all defects, which is the task ③ in Figure 1.

We now go though the whole process in detail taking the DNN architecture shown in Figure 3 as a running example.

Task ①: Potential-Defect Detection via Static Analysis. The DNN static analysis framework within RANUM first computes the numerical intervals of possible input and outputs for all nodes within the given DNN architecture, and then flag any nodes whose input intervals overlap with their invalid ranges as nodes with potential numerical defects.

[2] Bolded alphabet stands for vectors or tensors throughout the paper.

[3] Some architectures contain stochastic nodes. We view these nodes as nodes with randomly sampled data, so the architecture itself is deterministic.

In Figure 3, suppose that the user-specified input `x-input` (node 1) is within (elementwise, same below) range $[(-10,-10)^\intercal,(10,10)^\intercal]$; `weights` (node 2) is within range $[\begin{bmatrix}-10 & -10\\ -10 & -10\end{bmatrix},\begin{bmatrix}10 & 10\\ 10 & 10\end{bmatrix}]$; and `biases` (node 4) is within range $[(-10,-10)^\intercal,(10,10)^\intercal]$. Our DNN static analysis framework will compute out these interval abstractions for node inputs:

1) Node 5 (after `MatMul`): $[(-200,-200)^\intercal,(200,200)^\intercal]$;
2) Node 6 (after `Add`): $[(-210,-210)^\intercal,(210,210)^\intercal]$;
3) Node 8 (after `Softmax` in `float32`): $[(0,0)^\intercal,(1,1)^\intercal]$;
4) Node 9 (after `Sub` of $[1,1]$ and node 8), 10: $[(0,0)^\intercal,(1,1)^\intercal]$.

Since node 9 and 10 use `Log` operator whose invalid input range $(-\infty,U_{\min})$ overlaps their input range $[(0,0)^\intercal,(1,1)^\intercal]$, we flag node 9 and 10 as potential numerical defects.

This static analysis process follows the state-of-the-art DEBAR tool [59]. However, we extend DEBAR with a novel technique, backward fine-grained node labelling. This technique detects all nodes that require fine-grain abstractions, e.g., nodes that determine the control flow in a dynamic graph. For these nodes, we apply concrete execution instead of interval abstraction to prevent control flow ambiguity. As a result, the static analysis in RANUM supports much more DNN operators including dynamic control-flow operators like `Loop`.

Task ②: Feasibility Confirmation via Two-Step Test Generation. Given nodes that contain potential numerical defects (node 9 and 10 in our example), we generate failure-exhibiting system tests to confirm their feasibility. A failure-exhibiting system test is a tuple $\langle \boldsymbol{x}_{\mathsf{train}},\boldsymbol{x}_{\mathsf{infer}}\rangle$, such that after training the architecture with the training example $\boldsymbol{x}_{\mathsf{train}}$, with the trained model weights $\boldsymbol{w}_{\mathsf{infer}}$, the inference input $\boldsymbol{x}_{\mathsf{infer}}$ triggers numerical failure. The name "system test" is inspired from traditional software testing, where we test the method sequence (`m = train(`$\boldsymbol{x}_{\mathsf{train}}$`); m.infer(`$\boldsymbol{x}_{\mathsf{infer}}$`)`). In contrast, GRIST [52] generates model weights $\boldsymbol{w}_{\mathsf{infer}}$ and inference input $\boldsymbol{x}_{\mathsf{infer}}$ that only tests the inference method `m.infer()` and the weights may be infeasible from training. Hence, we view the GRIST-generated tuple $\langle \boldsymbol{w}_{\mathsf{infer}},\boldsymbol{x}_{\mathsf{infer}}\rangle$ as a "unit test".

We propose a two-step test generation technique to generate failure-exhibiting system tests.

***Step a**: Generate failure-exhibiting unit test $\langle \boldsymbol{w}_{\mathsf{infer}},\boldsymbol{x}_{\mathsf{infer}}\rangle$.* The state-of-the-art GRIST tool supports this step. However, GRIST solely relies on gradient back-propagation, which is relatively inefficient. In RANUM, we augment GRIST by combining its gradient back-propagation with random initialization inspired by recent DNN adversarial attack research [9, 22]. As a result, RANUM achieves 17.32X speedup with 100% success rate. Back to the running example in Figure 3, our RANUM can generate $\begin{bmatrix}5 & -5\\ -5 & 5\end{bmatrix}$ for node 2 and $(0.9,-0.9)^\intercal$ for node 4 as model weights $\boldsymbol{w}_{\mathsf{infer}}$; and $(10,-10)^\intercal$ for node 1 and $(1,0)^\intercal$ for node 11 as the inference input $\boldsymbol{x}_{\mathsf{infer}}$. Such $\boldsymbol{w}_{\mathsf{infer}}$ and $\boldsymbol{x}_{\mathsf{infer}}$ induce input $(0,1)^\intercal$ and $(1,0)^\intercal$ for node 9 and 10 respectively, which trigger numerical failures in both nodes.

***Step b**: Generate training example $\boldsymbol{x}_{\mathsf{train}}$ that achieves model weights $\boldsymbol{w}_{\mathsf{infer}}$.* To the best of our knowledge, there is no automatic approach for this task yet. RANUM provides support for this task based on our extension of DLG attack [61]. The DLG attack is originally designed for recovering the training data from training-phase gradient leakage. Here, we figure out the required training gradients to trigger the numerical failure at the inference phase and then leverage the DLG attack to generate $\boldsymbol{x}_{\mathsf{train}}$ that leads to such training gradient. Specifically, many DNN architectures contain operators such as `ReLU` on which DLG attack is hard to operate [34]. We combine straight-through estimator [3] to provide proxy gradients and bypass this barrier. Back to the running example in Figure 3, suppose the initial weights are $\begin{bmatrix}-0.1 & 0.1\\ 0.1 & -0.1\end{bmatrix}$ for node 2 and $(0,0)^\intercal$ for node 4, our RANUM can generate training example $\boldsymbol{x}_{\mathsf{train}}$ composed of $(5.635,-5.635)^\intercal$ for node 1 and $(1,0)^\intercal$ for node 11, such that one-step training with learning rate 1 on this example leads to $\boldsymbol{w}_{\mathsf{infer}}$. Combining $\boldsymbol{x}_{\mathsf{train}}$ from this step with $\boldsymbol{x}_{\mathsf{infer}}$ from *step a*, we obtain a failure-exhibiting system test that confirms the feasibility of potential defects in node 9 and 10.

Task ③: Fix Suggestion via Abstract Optimization. In this task, we suggest fixes for the confirmed numerical defects. RANUM is the first approach for this task to our knowledge.

The user may prefer different fix locations, which correspond to a user-specified set of nodes $\mathcal{V}_{\mathsf{fix}}\subseteq\mathcal{V}$ to impose the fix. For example, if the fix method is clipping the inference input, $\mathcal{V}_{\mathsf{fix}}$ are input nodes (e.g., nodes 1, 11 in Figure 3); if the fix method is clipping the model weights during training, $\mathcal{V}_{\mathsf{fix}}$ are weight nodes (e.g., nodes 2, 4 in Figure 3); if the fix method is clipping before the suspicious operator, $\mathcal{V}_{\mathsf{fix}}$ are nodes with numerical defects (e.g., nodes 9, 10 in Figure 3).

According to the empirical study of developers' fixes in Section I, 12 out of 13 defects are fixed by imposing interval preconditions clipping the inputs of $\mathcal{V}_{\mathsf{fix}}$. Hence, we suggest interval precondition as defect fix in this paper, which is interval constraint $\boldsymbol{l}_n \le f_n^{\mathsf{in}}(\boldsymbol{x};\boldsymbol{w}) \le \boldsymbol{u}_n$ for nodes $n\in\mathcal{V}_{\mathsf{fix}}$. A fix should satisfy that, when these constraints $\bigwedge_{n\in\mathcal{V}_{\mathsf{fix}}}(\boldsymbol{l}_n \le f_n^{\mathsf{in}}(\boldsymbol{x};\boldsymbol{w}) \le \boldsymbol{u}_n)$ are imposed, the input of any node in the computational graph should always be valid, i.e., $f_{n_0}^{\mathsf{in}}(\boldsymbol{x};\boldsymbol{w})\notin\mathcal{I}_{n_0,\mathsf{invalid}},\forall n_0\in\mathcal{V}$.

In RANUM, we formulate the fix suggestion task as a constrained optimization problem, taking the endpoints of interval abstractions for nodes in $\mathcal{V}_{\mathsf{fix}}$ as optimizable variables. We then propose the novel technique of abstraction optimization to solve this constrained optimization problem. Back to Figure 3 example, if users plan to impose fix on inference input, RANUM can suggest fix $-1\le$ `x-input` ≤ 1; if user plan to impose fix on nodes with numerical defects, RANUM can suggest fix $10^{-38}\le$ node 9 & node 10`.input` $\le +\infty$.

III. The RANUM Approach

In this section, we introduce the three novel techniques in RANUM: backward fine-grained node labelling in Section III-A; two-step test generalization in Section III-B; and abstraction optimization in Section III-C.

A. DNN Static Analysis Framework with Backward Fine-Grained Node Labelling for Potential-Defect Detection

RANUM contains a static analysis framework to enable potential-defect detection and support downstream tasks as shown in Figure 4. Given a DNN architecture and valid ranges for input and weight nodes, the static analysis framework computes interval abstractions for possible inputs and outputs of each node. As a result, we can check whether overlap exists between the interval abstraction and invalid input ranges for all nodes in the graph to detect potential numerical defects. Then, the defect nodes are fed into two-step test generation component to confirm feasibility of potential defects; and the differentiable abstractions are fed into abstract optimization component to produce fixes.

Formally, for given valid ranges of inference input and model weights, namely $\mathcal{X}$ and $\mathcal{W}$, for each node $n \in \mathcal{V}$, our framework will compute input interval abstraction $[\boldsymbol{l}_n, \boldsymbol{u}_n] := \{\boldsymbol{x} : \boldsymbol{l}_n \leq \boldsymbol{x} \leq \boldsymbol{u}_n\}$ such that $[\boldsymbol{l}_n, \boldsymbol{u}_n]$ captures all possible inputs of the node: $[\boldsymbol{l}_n, \boldsymbol{u}_n] \supseteq \{f_n^{\mathsf{in}}(\boldsymbol{x}, \boldsymbol{w}) : \boldsymbol{x} \in \mathcal{X}, \boldsymbol{w} \in \mathcal{W}\}$. We also compute output interval abstractions similarly.

Compared with traditional analysis tools for numerical software [11, 38], RANUM static analysis framework designs abstractions for DNN primitives operating on multi-dimensional tensors that are not supported by traditional tools. Compared with the state-of-the-art DEBAR tool [59], RANUM uses the same abstraction domain (interval domain with tensor partitioning), but incorporates a novel technique (backward fine-grained node labelling) to improve abstraction precision and support a wider range of DNN architectures.

Abstract Domain: Interval with Tensor Partitioning. Following DEBAR's design, we use the interval with tensor partitioning [59] as the abstraction domain, which partitions the tensor into several subblocks and shares the interval abstractions at the block level instead of imposing at the element level. Therefore, we can compute the abstraction of a smaller size than the original tensor to improve efficiency.

Our Technique: Backward Fine-Grained Node Labelling. The interval domain with tensor partitioning provides a degree of freedom in terms of the partition granularity, i.e., we can choose the subblock size for each node's abstraction. When choosing the finest granularity (i.e., elementwise abstraction) along with the initial data $\boldsymbol{x}$ for abstraction initialization, the abstraction interval $[\boldsymbol{x}, \boldsymbol{x}]$ contains only one element and corresponds to concrete execution. When choosing the coarsest granularity (i.e., one scalar to summarize the node tensor), the abstraction saves the most space but loses much precision. In DEBAR, the coarsest granularity is used by default for most operators. However, we find that using the finest instead of the coarsest granularity for some nodes is more beneficial for overall abstraction preciseness. For example, the control-flow operators (e.g., `Loop`) need concrete execution to determine the exact control flow in the dynamic graph, and the indexing operators (e.g., `Slice`) need concrete execution so we can precisely analyze other nodes. Hence, we propose to use the finest granularity, i.e., concrete execution, for some nodes (namely fine-grained requiring operators) while the coarsest granularity for other nodes during static analysis.

To allow concrete execution for required nodes, typically, all of their preceding nodes also need concrete execution. Otherwise, the uncertainty intervals from preceding nodes will be propagated till these nodes, resulting in precision loss. To solve this problem, in RANUM, we back-propagate "fine-grain" labels from these fine-grain requiring nodes to initial nodes by topologically sorting the graph with *inverted* edges, and then apply the finest granularity abstractions on all labelled nodes. In practice, we find this strategy eliminates the control-flow ambiguity and indexing ambiguity with little loss of efficiency. As a result, RANUM supports all dynamic graphs (which are not supported by DEBAR) that compose of 39.2% of the benchmarks proposed by Yan et al. [52].

Furthermore, when preceding nodes use finer-grain abstraction granularity, the subsequent nodes should preserve such fine granularity to preserve the analysis preciseness and the concrete execution flow. Principally, the choice of abstraction granularity should satisfy both tightness (bears no precision loss compared to elementwise interval abstraction) and minimality (uses the minimum number of partitions for high efficiency). To realize these principles, we dynamically determine node's abstraction granularity based on the granularity of preceding nodes. The abstraction design for some operators is non-trivial. Detail formulation, illustration, proofs, and documentation of some non-trivial operators are in Suppl. C.

In summary, the whole static analysis process consists of three steps: (1) Determine the tensor partition granularity of all initial nodes by our technique of backward fine-grained node labelling. (2) Sort all nodes in the graph in topological order. (3) Apply corresponding abstraction computation algorithms for each node based on preceding node's abstractions.

B. Two-Step Test Generation for Feasibility Confirmation

RANUM generate failure-exhibiting system tests for the given DNN to confirm the feasibility of potential numerical defects. Here, we take the DNN architecture as the input. From the static analysis framework, we obtain a list of nodes that have potential numerical defects. For each node n_0 within the list, we apply our technique of two-step test generation to produce a failure-exhibiting system test $t_{\mathsf{sys}} = \langle \boldsymbol{x}_{\mathsf{train}}, \boldsymbol{x}_{\mathsf{infer}} \rangle$ as the output. According to Section II-B, the test should satisfy that after the architecture is trained with $\boldsymbol{x}_{\mathsf{train}}$, entering $\boldsymbol{x}_{\mathsf{infer}}$ in the inference phase results in numerical failure.

We propose the novel technique of two-step test generalization: First, generate failure-exhibiting unit test $\langle \boldsymbol{w}_{\mathsf{infer}}, \boldsymbol{x}_{\mathsf{infer}} \rangle$; Then, generate training example $\boldsymbol{x}_{\mathsf{train}}$ that leads model weights to be close to $\boldsymbol{w}_{\mathsf{infer}}$ after training.

Step a: Unit Test Generation. As sketched in Section II-B, we strengthen the state-of-the-art unit test generation approach, GRIST [52], by combining with random initialization to complete this step. Specifically, GRIST leverages the gradients of the defect node's input with respect to the inference input and weights to iteratively update the inference input and weights to generate failure-exhibiting unit tests. However, GRIST always conducts updates from existing inference input and weights, which can suffer from local minima problem [22]. Instead, motivated by DNN adversarial attack literature [22, 43], a sufficient number of random starts helps to find global minima effectively. Hence, in RANUM, we first conduct uniform sampling for 100 times for both the inference input and weights to trigger the numerical failure. If no failure is triggered, we use the sample that induces the smallest loss as the start point for gradient optimization. As Section V-A shows, this strategy substantially boosts the efficiency, achieving 17.32X speed-up.

Step b: Training Example Generation. For this step, RANUM takes the following inputs: (1) the DNN architecture; (2) the failure-exhibiting unit test $t_{\mathsf{unit}} = \langle \boldsymbol{w}_{\mathsf{infer}}, \boldsymbol{x}_{\mathsf{infer}} \rangle$; and (3) the randomly initialized weights $\boldsymbol{w}_0$. Our goal is to generate a legal training example $\boldsymbol{x}_{\mathsf{train}}$, such that the model trained with $\boldsymbol{x}_{\mathsf{train}}$ will contain weights close to $\boldsymbol{w}_{\mathsf{infer}}$.

DNNs are typically trained with gradient-descent-based algorithms such as stochastic gradient descent (SGD). In SGD, in each step t, we sample a mini-batch of samples from the training dataset to compute their gradients on model weights and use them to update the weights. We focus on one-step SGD training with a single training example, since generating a single one-step training example to exhibit failure is more desirable for ease of debugging as discussed in Section I. In this case, after training, the model weights

$$\boldsymbol{w}_{\mathsf{infer}} = \boldsymbol{w}_0 - \gamma \nabla_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{x}_{\mathsf{train}}; \boldsymbol{w}_0), \tag{1}$$

where $\gamma \in \mathbb{R}_+$ is a predefined learning rate, and $\mathcal{L}$ is the predefined loss function in the DNN architecture. Hence, our goal becomes finding $\boldsymbol{x}_{\mathsf{train}}$ that satisfies

$$\nabla_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{x}_{\mathsf{train}}; \boldsymbol{w}_0) = (\boldsymbol{w}_0 - \boldsymbol{w}_{\mathsf{infer}})/\gamma. \tag{2}$$

The DLG attack (deep leakage from gradient, [61]) is a method for generating input data that induce specific weight gradients. The attack is originally designed for recovering training samples from monitored gradient updates. Since the right hand side (RHS) of Equation (2) is known, our goal here is also generating input example $\boldsymbol{x}_{\mathsf{train}}$ that induces specific weight gradients. Therefore, we leverage DLG attack to generate training example $\boldsymbol{x}_{\mathsf{train}}$.

Extending DLG Attack with Straight-Through Estimator. Directly using DLG attack suffers from an optimization challenge in our scenario. Specifically, in DLG attack, suppose the target weight gradients are $\Delta \boldsymbol{w}_{\mathsf{targ}}$, we use gradient descent over the squared error $\|\nabla_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{x}; \boldsymbol{w}_0) - \Delta \boldsymbol{w}_{\mathsf{targ}}\|_2^2$ to generate $\boldsymbol{x}$. In this process, we need meaningful gradient information of this squared error loss to perform the optimization. However, the gradient of this loss involves second-order derivatives of $\mathcal{L}(\boldsymbol{x}; \boldsymbol{w}_0)$, which could be zero. For example, DNNs with ReLU as activation function are piecewise linear and have zero second-order derivatives almost everywhere [34]. This optimization challenge is partly addressed in DLG attack by replacing `ReLU` with `Sigmoid`, but it changes the DNN architecture (i.e., the system under test) and hence not suitable.

We leverage the straight-through estimator to mitigate the optimization challenge. Specifically, for some operators, such as `ReLU`, we do not change its forward computation, but change its backward gradient computation to provide second-order derivatives within the DLG attack process. For example, for `ReLU`, in backward computation we use the gradient of $\mathrm{Softplus}$ function, namely $1 - \frac{1}{1+\exp(\boldsymbol{x})}$, because $\mathrm{Softplus}$ is a smoothed approximation of ReLU function [8] with non-zero second-order derivatives. Note that we modify the computed gradients only within the DLG attack process. After such $\boldsymbol{x}_{\mathsf{train}}$ is generated by the DLG attack, we evaluate whether it triggers numerical failure using the original architecture and original gradients in Equation (1).

C. Abstraction Optimization for Fix Suggestion

In this task, we aim to generate the precondition fix given imposing locations. The inputs are the DNN architecture, the node n_0 with numerical defects, and a node set $\mathcal{V}_{\mathsf{fix}}$ to impose the fix. We would like to generate interval preconditions for $\mathcal{V}_{\mathsf{fix}}$ node inputs, so that after these preconditions are imposed, the defect on n_0 is fixed.

Formally, our task is to figure out $\langle l_n, u_n \rangle$ for each $n \in \mathcal{V}_{\mathsf{fix}}$ (l_n and u_n are scalars so the same interval bound applied to all elements of n's tensor), such that for any $\boldsymbol{x}, \boldsymbol{w}$ satisfying $f_n^{\mathsf{in}}(\boldsymbol{x}; \boldsymbol{w}) \in [l_n, u_n]$, $\forall n \in \mathcal{V}_{\mathsf{fix}}$, for the defect node n_0, we have $f_{n_0}^{\mathsf{in}}(\boldsymbol{x}; \boldsymbol{w}) \notin \mathcal{I}_{n_0,\mathsf{invalid}}$, where the full list of invalid input ranges $\mathcal{I}_{n_0,\mathsf{invalid}}$ is in Suppl. B. We formulate a surrogate optimization problem for this task as below.

$$\underset{l_n, u_n : n \in \mathcal{V}_{\mathsf{fix}}}{\text{maximize}} \quad s \tag{3}$$

$$\text{s.t.} \quad u_n \ge l_n + s(u_n^{\mathsf{valid}} - l_n^{\mathsf{valid}}), \forall n \in \mathcal{V}_{\mathsf{fix}}, \tag{4}$$

$$l_n^{\mathsf{valid}} \le l_n \le u_n \le u_n^{\mathsf{valid}}, \forall n \in \mathcal{V}_{\mathsf{fix}}, \tag{5}$$

$$\mathcal{L}_{n_0}^{\mathsf{precond}}(\{l_n, u_n\}_{n \in \mathcal{V}_{\mathsf{fix}}}) < 0. \tag{6}$$

Here, l_n^{valid} and u_n^{valid} are the valid ranges of the node's input n, which is fixed and determined by the valid ranges of input and weights. $\mathcal{L}_{n_0}^{\mathsf{precond}}$ is the node-specific precondition generation loss that is the distance between the furthest endpoint of defect node n_0's interval abstraction and n_0's valid input range. Hence, when $\mathcal{L}_{n_0}^{\mathsf{precond}}(\{l_n, u_n\}_{n \in \mathcal{V}_{\mathsf{fix}}})$ becomes negative, the solution $\{l_n, u_n\}_{n \in \mathcal{V}_{\mathsf{fix}}}$ is a valid precondition. The optimization variables are the precondition interval endpoints l_n and u_n and the objective is the relative span of these intervals. The larger the span is, the looser the precondition constraints are, and the less hurt they are for the model's utility. Equation (4) enforces the interval span requirement. Equation (5) assures that the precondition interval is in valid range. Equation (6) guarantees the validness of the precondition as a fix.

For any $\{l_n, u_n\}_{n \in \mathcal{V}_{\mathsf{fix}}}$, thanks to RANUM static analysis framework, we can compute induced intervals of defect node n_0, and thus compute the loss value $\mathcal{L}_{n_0}^{\mathsf{precond}}$.

As shown in Algorithm 1, we propose the technique of **abstraction optimization** to effectively and approximately solve this optimization. Our technique works iteratively. In the first iteration, we set span $s = 1$, and in the subsequent iterations, we reduce the span s exponentially as shown in line 13 where hyperparameter $\gamma_s = 0.9$. Inside each iteration, for each node to impose precondition $n \in \mathcal{V}_{\mathsf{fix}}$, we use the interval center $c_n = (l_n + u_n)/2$ as the optimizable variable and compute the *sign* of its gradient: $\mathrm{sgn}(\nabla_{c_n} \mathsf{loss})$. We use

Algorithm 1 Abstraction Optimization (Section III-C)

```
Input: DNN architecture $\mathcal{G} = \langle \mathcal{V}, \mathcal{E} \rangle$, defect node $n_0 \in \mathcal{V}$, nodes to
    impose fix $\mathcal{V}_{\textsf{fix}} \subseteq \mathcal{V}$
 1: $s \leftarrow 1, \gamma_s \leftarrow 0.9, \gamma_c \leftarrow 0.1, \textsf{minstep} \leftarrow 0.1, \textsf{maxiter} \leftarrow 1000$
 2: $c_n \leftarrow (l_n^{\textsf{valid}} + u_n^{\textsf{valid}})/2, l_n \leftarrow l_n^{\textsf{valid}}, u_n \leftarrow u_n^{\textsf{valid}}, \forall n \in \mathcal{V}_{\textsf{fix}}$
 3: for $i = 1$ to $\textsf{maxiter}$ do
 4:     for $n \in \mathcal{V}_{\textsf{fix}}$ do
 5:         $\textsf{loss} \leftarrow \mathcal{L}_{n_0}^{\textsf{precond}}(\{l_{n'}, u_{n'}\}_{n' \in \mathcal{V}_{\textsf{fix}}})$
 6:         $c_n \leftarrow c_n - \gamma_c \max\{|c_n|, \textsf{minstep}\} \mathrm{sgn}(\nabla_{c_n} \textsf{loss})$
 7:         $(l_n, u_n) \leftarrow (c_n - \frac{s(u_n^{\textsf{valid}} - l_n^{\textsf{valid}})}{2}, c_n + \frac{s(u_n^{\textsf{valid}} - l_n^{\textsf{valid}})}{2})$
 8:         $(l_n, u_n) \leftarrow (\max\{l_n, l_n^{\textsf{valid}}\}, \min\{u_n, u_n^{\textsf{valid}}\})$
 9:     end for
10:     if $\mathcal{L}_{n_0}^{\textsf{precond}}(\{l_n, u_n\}_{n \in \mathcal{V}_{\textsf{fix}}}) < 0$ then
11:         return $\{l_n, u_n\}_{n \in \mathcal{V}_{\textsf{fix}}}$                // Find precondition fix
12:     end if
13:     $s \leftarrow \gamma_s \cdot s$
14: end for
15: return "failed"                          // Failed to find precondition fix
```

this gradient sign to update each c_n towards the direction of reducing loss value in line 6. Then, we use c_n and the span s to recover the actual interval in line 7 and clip l_n and u_n by the valid range $[l_n^{\textsf{valid}}, u_n^{\textsf{valid}}]$ in line 8. At the end of this iteration, for updated l_n and u_n, we compute $\mathcal{L}_{n_0}^{\textsf{precond}}(\{l_n, u_n\}_{n \in \mathcal{V}_{\textsf{fix}}})$ to check whether the precondition is a fix. If so, we terminate; otherwise, we proceed to the next iteration.

Remark. The key ingredient in the technique is the gradient-sign-based update rule shown in line 6 that is much more effective than normal gradient descent due to following two reasons. (1) Our update rule can get rid of gradient explosion and vanishing problem. For early optimization iterations, the span s is large and interval bounds are generally coarse, which results in too large or too small gradient magnitude. For example, the input range for `Log` could be $[1, 10^{10}]$ where gradient can be 10^{-10}, resulting in almost negligible gradient updates. In contrast, our update rule leverages the sign of gradient which always points to the correct gradient direction. The update step size in our rule is the maximum of current magnitude $|c_n|$ and $\textsf{minstep}$ to avoid stagnation. (2) Our update rule mitigates the gradient magnitude discrepancy of different c_n. At different locations, the nodes in DNNs can have diverse value magnitudes and these magnitudes are not aligned with their gradient magnitudes, which makes the gradient optimization challenging. Therefore, we use this update rule to solve the challenge, where the update magnitude depends on the *value* magnitude ($|c_n|$) instead of *gradient* magnitude ($\nabla_{c_n} \textsf{loss}$). We empirically compare our technique with standard gradient descent in Section V-C.

IV. Implementation

We implement a tool for RANUM in roughly 10k lines of Python code based on PyTorch. Our tool leverages existing modules (`tf2onnx` for PyTorch and `torch.onnx.export` for Tensorflow and Keras) to extract ONNX-format DNN architectures from DL programs to take as the input, and automatically generates detection result, system tests, and defect fixes (given fix imposing location).

V. Experimental Evaluation

We conduct a systematic experimental evaluation to answer the following research questions.

RQ1 For tasks already supported by existing state-of-the-art (SOTA) tools (task ① and ②a), how much more effective and efficient is RANUM compared to these SOTA tools?

RQ2 For feasibility confirmation via *generating failure-exhibiting system tests* (task ②), how much more effectively and efficiently can RANUM confirm potential numerical defects compared to baseline approaches?

RQ3 For *suggesting fixes* (task ③), how much more efficient and effective is RANUM in terms of guarding against numerical failures compared to baseline approaches and developers' fixes, respectively?

For RQ1, we compare RANUM with all SOTA tools. For RQ2 and RQ3, RANUM is the first approach to the best of our knowledge, so we compare RANUM with baseline approaches (constructed by leaving our novel techniques out of RANUM) and developers' fixes. We conduct the evaluation on the GRIST benchmarks [52], being the largest dataset of real-world DNN numerical defects to our knowledge. The benchmarks contain 63 real-world DL programs with numerical defects collected from previous studies and GitHub. Each program contains a DNN architecture, and each architecture has one or more numerical defects. There are 79 true numerical defects in total.

We evaluate all approaches on the same workstation with 24-core Xeon E5-2650 CPU running at 2.20 GHz and one GTX 1080 Ti GPU. Throughout the evaluation, we stop the execution after reaching 30 min limit by following the evaluation setup by the most recent related work [52].

A. RQ1: Comparison with SOTA Tools

As discussed in Section II-B, for two tasks, existing tools can provide automatic support: potential-defect detection (task ①) where the SOTA tool is DEBAR [59], and failure-exhibiting unit test generation (task ②a) where the SOTA tool is GRIST [52]. We compare RANUM with these tools on their supported tasks respectively.

Comparison with DEBAR. RANUM successfully detect all 79 true defects and DEBAR only detects 48 true defects according to both our evaluation and the literature [52]. Hence, RANUM detects 64.58% more true defects than DEBAR. In terms of efficiency, DEBAR and RANUM have similar running time and both finish in 3 s per case.

We manually inspect the cases where DEBAR fails but RANUM succeeds. They correspond to DL programs written with PyTorch library, which generates dynamic computational graphs that DEBAR cannot handle. In contrast, RANUM provides effective static analysis support for dynamic computational graphs thanks to our backward fine-grained node labelling technique introduced in Section III-A that is capable of determining the control flow within dynamic graphs.

Comparison with GRIST. Results are shown in Table I. Since both RANUM and GRIST has randomness component

TABLE I

(RQ1) RESULTS OF TASK ②A (FAILURE-EXHIBITING **UNIT** TEST GENERATION) WITH RANUM AND GRIST [52]. C IS THE TOTAL NUMBER OF TIMES THAT NUMERICAL FAILURES ARE TRIGGERED IN 10 REPEATED RUNS, T REFERS TO THE AVERAGE EXECUTION TIME PER RUN, AND ⇑T IS THE AVERAGE TIME IMPROVEMENT ACHIEVED BY RANUM COMPARED TO GRIST.

Case	RANUM			GRIST		Case	RANUM			GRIST		Case	RANUM			GRIST		Case	RANUM			GRIST	
ID	C	T	⇑T	C	T	ID	C	T	⇑T	C	T	ID	C	T	⇑T	C	T	ID	C	T	⇑T	C	T
1	10	9.01	1.20 X	10	10.77	16b	10	0.21	20.85 X	10	4.42	32	10	0.06	27.93 X	10	1.77	47	10	0.06	32.51 X	10	1.87
2a	10	0.02	9.75 X	10	0.24	16c	10	0.25	17.54 X	10	4.43	33	10	0.06	33.63 X	10	1.91	48a	10	0.38	3.06 X	10	1.17
2b	10	0.03	614.68 X	10	16.54	17	10	439.19	$+\infty$	0	-	34	10	0.06	33.95 X	10	1.90	48b	10	0.15	7.12 X	10	1.10
3	10	0.02	432.11 X	10	8.67	18	10	0.02	1040.46 X	10	22.17	35a	10	0.44	61.76 X	10	27.33	49a	10	0.49	41.09 X	10	20.07
4	10	0.01	1.00 X	10	0.01	19	10	0.16	689.66 X	10	107.78	35b	10	0.45	819.02 X	10	364.86	49b	10	0.50	612.22 X	10	307.24
5	10	0.05	6.48 X	10	0.34	20	10	0.16	3237.27 X	10	511.06	36a	10	0.44	41.80 X	10	18.58	50	10	0.16	781.02 X	10	126.80
6	10	0.84	5.20 X	10	4.38	21	10	0.16	259.73 X	10	42.09	36b	10	0.46	783.16 X	10	362.41	51	10	1.88	671.55 X	3	1263.04
7	10	0.87	4.54 X	10	3.96	22	10	0.94	1518.43 X	10	1433.12	37	10	0.06	38.34 X	10	2.39	52	10	0.15	336.37 X	10	50.59
8	10	0.86	4.63 X	10	3.99	23	10	0.01	157.72 X	10	1.88	38	10	0.06	34.50 X	10	1.94	53	10	0.05	36.64 X	10	1.92
9a	10	0.20	11.03 X	10	2.22	24	10	0.81	40.72 X	10	33.05	39a	10	0.43	42.79 X	10	18.30	54	10	0.05	36.76 X	10	1.83
9b	10	0.14	14.46 X	10	2.09	25	10	0.04	1271.88 X	10	44.70	39b	10	0.43	843.66 X	10	362.22	55	10	0.82	44.63 X	10	36.63
10	10	0.17	228.42 X	10	39.64	26	10	0.05	37.96 X	10	2.00	40	10	0.04	1995.27 X	10	85.97	56	10	0.06	35.04 X	10	1.93
11a	10	0.15	27.58 X	10	4.26	27	10	0.01	185.61 X	10	1.91	41	10	0.04	1967.23 X	10	86.36	57	10	0.01	177.45 X	10	1.88
11b	10	0.13	34.75 X	10	4.38	28a	10	24.37	-13.30 X	10	1.83	42	10	0.05	1934.84 X	10	87.89	58	10	0.83	12.01 X	10	9.95
11c	10	0.11	4499.86 X	10	516.13	28b	10	24.17	7.28 X	10	176.02	43a	10	0.48	35.63 X	10	16.96	59	10	0.02	105.40 X	10	1.94
12	10	0.26	135.94 X	10	34.69	28c	10	0.12	8.69 X	10	1.02	43b	10	0.45	4008.93 X	10	1800.00	60	10	0.15	221.97 X	10	34.19
13	10	0.01	1.10 X	10	0.01	28d	10	0.12	1518.28 X	10	176.02	44	10	0.27	579.29 X	10	155.38	61	10	0.35	53.29 X	10	18.78
14	10	0.80	107.96 X	10	86.23	29	10	0.89	16.83 X	10	14.98	45a	10	0.16	417.25 X	10	68.08	62	10	1.85	72.19 X	10	133.62
15	10	1.71	5.95 X	10	10.18	30	10	0.16	222.12 X	10	35.61	45b	10	0.88	14.69 X	10	12.98	63	10	2.06	117.12 X	10	240.68
16a	10	0.12	34.24 X	10	4.02	31	10	3.13	2.41 X	3	7.54	46	10	0.01	168.39 X	10	1.88	**Tot: 79**	**790**	**6.66**	**17.32 X**	**766**	**115.30**

TABLE II

(RQ2) RESULTS OF TASK ② (FAILURE-EXHIBITING **SYSTEM** TEST GENERATION) WITH RANUM AND RANDOM (BASELINE). C IS THE TOTAL NUMBER OF TIMES THAT NUMERICAL FAILURES ARE TRIGGERED IN 10 REPEATED RUNS, AND T REFERS TO THE AVERAGE EXECUTION TIME PER RUN.

Case	RANUM		Random		Case	RANUM		Random		Case	RANUM		Random		Case	RANUM		Random		Case	RANUM		Random	
ID	C	T	C	T	ID	C	T	C	T	ID	C	T	C	T	ID	C	T	C	T	ID	C	T	C	T
1	0	9.01	0	1806.13	13	10	0.01	10	0.01	27	10	0.01	10	0.01	38	1	0.13	0	1800.65	49b	10	0.50	8	364.09
2a	10	0.03	10	0.06	14	10	12.60	10	0.50	28a	0	24.37	0	1920.29	39a	10	0.43	1	1623.72	50	10	4.89	10	0.16
2b	10	0.03	10	0.06	15	0	1.71	0	2107.24	28b	0	24.17	0	1911.26	39b	10	0.43	8	364.10	51	10	49.12	10	2.10
3	10	0.02	10	0.05	16a	10	0.12	10	0.75	28c	10	0.12	10	0.53	40	10	0.06	10	0.02	52	10	4.87	10	0.15
4	10	0.01	10	0.01	16b	10	0.21	0	1834.44	28d	10	0.12	10	0.48	41	10	0.06	10	0.02	53	10	0.07	10	0.03
5	10	0.05	10	0.06	16c	10	0.25	0	1831.67	29	10	0.89	10	11.96	42	10	0.06	10	0.02	54	10	0.07	10	0.02
6	10	0.84	10	12.42	17	10	549.98	10	235.11	30	10	4.88	10	0.14	43a	10	0.48	1	1623.71	55	10	0.82	10	12.79
7	10	0.87	10	12.51	18	10	0.02	10	0.05	31	10	14.62	10	9.31	43b	10	0.45	9	184.14	56	10	0.07	10	0.03
8	10	0.86	10	12.37	19	10	4.88	10	0.16	32	10	0.08	10	0.03	44	10	0.27	10	1.36	57	10	0.01	8	360.01
9a	10	0.20	7	541.25	20	10	4.88	10	0.14	33	10	0.07	10	0.02	45a	10	4.89	10	0.15	58	10	0.83	10	12.28
9b	10	0.14	10	1.39	21	10	4.89	10	0.14	34	10	0.42	10	0.20	45b	10	0.88	10	12.27	59	10	0.02	10	0.05
10	10	4.90	10	0.16	22	10	500.10	0	1801.60	35a	10	0.44	10	4.01	46	10	0.01	10	0.01	60	10	4.88	10	0.16
11a	10	0.15	10	0.72	23	10	0.01	10	0.01	35b	10	0.45	10	4.22	47	10	0.08	10	0.03	61	10	9.84	10	0.93
11b	10	0.13	10	0.76	24	10	0.81	10	12.52	36a	10	0.44	1	1623.76	48a	10	9.89	10	0.90	62	10	48.86	10	2.73
11c	10	0.11	10	0.74	25	10	0.04	10	0.15	36b	10	0.46	3	1263.79	48b	10	4.88	10	0.15	63	10	49.06	10	2.15
12	10	0.26	10	0.72	26	10	0.07	10	0.03	37	2	0.07	2	1440.50	49a	10	0.49	1	1623.88	**Tot: 79**	**733**	**17.31 (19.30X)**	**649**	**334.14**

where RANUM uses random initialization and GRIST relies on DNN's randomly initialized weights, we repeat both approaches for 10 runs, record the total number of times where failure-exhibiting unit test is generated, and the average execution time per run. We observe that RANUM succeeds in *all* cases and *all* repeated runs, and GRIST fails to generate unit test in 24 out of 790 runs (i.e., with 96.96% success rate). Moreover, RANUM has 6.66 s average execution time and is 17.32X faster than GRIST.

The superior effectiveness and efficiency of RANUM is largely due to the existence of random initialization as introduced in Section III-B. We observe that since GRIST always takes initial model weights and inference input as the start point to update from, the unit test is just slightly changed from initial weights and input, which can be hard to trigger the oblivious numerical defect. In contrast, RANUM uses random initialization to explore a much larger space and combines gradient-based optimization to locate the failure-exhibiting instances from the large space. We also evaluated the pure random strategy that only uses random initialization without gradient-based optimization, and such strategy fails in 30 runs, being inferior than both RANUM and GRIST, which means both random initialization and gradient-based optimization are important. Among all 79 cases, RANUM is slower than GRIST on only one case (28a). For this case, we find the default inference input loaded by the DNN program is not far from the failure-exhibiting one. In only this specific case, GRIST could be more efficient.

B. RQ2: Feasibility Confirmation via System Test Generation

In task ②, RANUM confirms feasibility of potential numerical defects by generating failure-exhibiting system tests.

Baseline. Since RANUM is the first approach for this task, we did not compare with existing literature and propose one random-based approach (called "Random" hereinafter) as the baseline. In "Random", we first generate failure-exhibiting unit test with random sampling. If there is any sample that triggers the failure, we stop and keep the inference example part as the desired $\boldsymbol{x}_{\mathsf{infer}}$. Then, we generate $\boldsymbol{x}_{\mathsf{train}}$ again by random sampling. If any sample, when used for training, could induce model weights $\boldsymbol{w}$ that cause numerical failure when using $\boldsymbol{x}_{\mathsf{infer}}$ as the inference input, we keep that sample as $\boldsymbol{x}_{\mathsf{train}}$ and terminate. If and only if both $\boldsymbol{x}_{\mathsf{infer}}$ and $\boldsymbol{x}_{\mathsf{train}}$ are found, we count this run as a "success" one for "Random".

For each defect, due to the randomness of model's initial weights, we repeat both RANUM and "Random" for 10 runs. Both approaches use the same set of random seeds.

Evaluation Result. Results are in Table II. We observe that RANUM succeeds in $733/(79 \times 10) = 92.78\%$ runs and the baseline "Random" succeeds in $649/(79 \times 10) = 82.15\%$ runs. Moreover, RANUM spends only 17.31 s time on average per run which is a 19.30X speed-up compared to "Random". We also observe that RANUM is more reliable across repeated runs. There are only 6 cases with unsuccessful repeated runs in RANUM, but there are 19 such cases in "Random". Hence, RANUM is substantially more effective, efficient, and reliable for generating system tests than the baseline.

TABLE III

(RQ3) Results of task ③ (fix suggestion) under three imposing location specifications with RANUM and two baselines (RANUM-E and GD). # counts number of fixes found. "Time (s)" is the total running time for all 79 cases.

Imposing	RANUM		RANUM-E		GD	
Locations	#	Time (s)	#	Time (s)	#	Time (s)
Weight + Input	**79**	**54.23**	78	540.13	57	188.63
Weight	**72**	**58.47**	71	581.86	43	219.28
Input	**37**	**924.74**	**37**	3977.30	29	952.19

Discussion. The high effectiveness of RANUM mainly comes from the advantage of gradient guided search compared with random search. As described in Section III-B, RANUM leverages both first-order gradients (in step a) and second-order derivatives (in step b) to guide the search of system tests. In contrast, "Random" uses random sampling hoping that failure-exhibiting training examples can reveal after sufficient sampling. Hence, when such training examples are relatively rare in the whole valid input space, "Random" is less effective. We conduct an ablation study in Suppl. E which shows RANUM improves over "Random" in both steps inside RANUM.

Failing Case Analysis. We studied all six defects where RANUM may fail. We found that (1) four defects (Case ID 1, 15, 37, and 38) are because the architecture is challenging for gradient-based optimization, e.g., due to `Min`/`Max`/`Softmax` operators which provides little or no gradient information. We leave it as future work to solve these cases, which may need to dynamically detect operators with vanishing gradients and reconstruct the gradient flow. (2) two defects (Case ID 28a and 28b) correspond to those that caused by `Div` operators where only close-to-zero divisor can trigger the numerical failure. Hence, for operators with narrow invalid ranges, RANUM may fail to generate failure-exhibiting training example.

C. RQ3: Fix Suggestion

At task ③, RANUM suggests fixes for numerical defects. We compare RANUM with fixes generated by baseline approaches and developers' fixes.

*1) **Comparison between RANUM and Baselines**:* RANUM is the first approach for this task, and we propose two baseline approaches to compare with. (1) "RANUM-E": this approach changes the abstraction domain of RANUM from interval with tensor partitioning to standard interval. (2) "GD": this approach uses standard gradient descent for optimization instead of the abstraction optimization technique in RANUM.

Evaluation Protocol. We evaluate whether each approach can generate fixes that eliminate *all* numerical defects for a given DNN architecture given imposing locations. We consider three types of locations: on both weight and input nodes, on only weight nodes, and on only input nodes. In practice, model providers can impose fixes on weight nodes by clipping weights after the model is trained; and users can impose fixes on input nodes by clipping their inputs before loading them into the model. Since all approaches are deterministic, for each case we run only once. We say that the fix eliminates all numerical defects if and only if (1) the RANUM static analysis framework cannot detect any defects from the fixed architecture; and (2) 1,000 random samples cannot trigger any numerical failures after imposing the fix.

Evaluation Result. We report the statistics, including number of successful cases among all 79 cases and the total running time, in Table III. From the table, we observe that on all the three imposing location settings, RANUM always succeeds on the most cases and spends much less time. For example, when fixes can be imposed on both weights and input nodes, RANUM succeeds on *all* cases with a total running time $54.23\,\mathrm{s}$. In contrast, RANUM-E requires $> 10\times$ time, and GD succeeds in only 72.15% cases. Hence, RANUM is substantially more effective and efficient for suggesting fixes compared to baseline approaches.

Discussion. The two baseline approaches can be viewed as ablated versions of RANUM. Comparing RANUM and GD, we conclude that the technique of abstraction optimization substantially improves the effectiveness and also helps the efficiency. Comparing RANUM and RANUM-E, we conclude that the interval abstraction with tensor partitioning as the abstraction domain substantially improves the efficiency and also helps the effectiveness.

We find that it is much easier to find the fix when the imposing locations are weight nodes compared to input nodes. For example, when imposing on weights, RANUM can succeed on 72 cases but on input it succeeds on only 37 cases. Since model providers can impose fixes on weights and users impose on inputs, this finding implies that fixing numerical defects on providers' side may be more effective than on end-users' side.

*2) **Comparison between RANUM and Developers' Fixes**:* We conduct an empirical study to compare the fixes generated by RANUM and by the developers.

Evaluation Protocol. We manually locate GitHub repositories from which the GRIST benchmarks are constructed. Among the 79 cases, we find the repositories for 53 cases on GitHub and we study these cases. We locate the developers' fixes of the numerical defects by looking at issues and follow-up pull requests. Then, two authors independently compare the developers' fixes and RANUM fixes and classify the comparison results. Since RANUM suggests different fixes for different imposing locations, for each case we first determine the imposing locations from the developer's fix, then compare with RANUM's fix for that location. Two authors coordinate their results and discuss misaligned results to reach consensus.

Results. We categorize the comparison results between RANUM precondition fixes and developers' fixes as below.

A *(30 cases) Better than developers' fixes or no available developer's fix.* Developers either propose no fix or use heuristic fixes, such as reducing the learning rate or using the mean value to reduce the variance. These fixes may work in practice but cannot rigorously guarantee the elimination of the numerical defect for any training or inference data. In contrast, RANUM generates better fixes since these fixes rigorously eliminate the defect.

B *(7 cases) Equivalent to developers' fixes.* Developers and RANUM suggest equivalent or highly similar fixes.

C *(13 cases) No need to fix.* For these cases, there is no need to fix the numerical defect in the given architecture. There are mainly three reasons. (1) The DNN is used in the whole project with fixed weights or test inputs. As a result, though the architecture contains defects, it will not lead to system failure. (2) The architecture is injected a defect as a test case for automatic tools, such as a test architecture in `TensorFuzz` [26] repository. (3) The defect can be hardly triggered in practice. For example, the defect is in a `Div` operator where the divisor can be very close to zero to trigger divide-by-zero failure, but this hardly happens in practice since the divisor is randomly initialized.

D *(3 cases) Inferior than developers' fixes or RANUM-generated fixes are impractical.* In two cases, RANUM-generated fixes are inferior than developers' fixes. Human developers observe that the defect operator is `Log`, and its input is non-negative. So they propose to add 10^{-6} to the input of `Log` as the fix. In contrast, RANUM can only generate a clipping-based fix, e.g., clipping the input if it is less than 10^{-6}, due to the nature of precondition. When the input is small, RANUM's fix interrupts the gradient flow from output to input while human's fix maintains it. As a result, human's fix does less hurt to model's trainability and is better than RANUM's fixes. In one other case, the RANUM-generated fix imposes a small span for some model weights (less than 0.1 for each component of that weight node). Such a small weight span strongly limits the model's expressiveness and capacity.

Discussion. From the comparison results, we can conclude that for the 40 cases where numerical defects are needed to be fixed (exclude case C), RANUM suggests equivalent or better fixes than human developers in 37 cases. Therefore, RANUM is comparably effective as human developers in terms of suggesting numerical defect fixes, and is much more efficient since RANUM is an automatic approach.

To further improve RANUM, two possible directions are (1) Considering more forms of fixes: RANUM only supports imposing interval preconditions as the fix. From human developers, there are more fix forms such as adding or subtracting a constant number or replacing the operator type [14]. For example, for a `Log` operator whose input is non-negative, we can suggest adding a constant positive value to input as a better fix than imposing interval precondition. (2) Tightening the static analysis: With a tighter (i.e., more precise) static analysis, we can synthesize a looser precondition as the fix.

VI. Threats to Validity

An external threat to validity is the evaluation subject. We evaluate RANUM and baseline approaches on the GRIST benchmarks [52]. Though the dataset contains 63 real-world DL programs and is the largest to our knowledge, it may still not be representative enough. Another external threat is the approach randomness. To mitigate this threat, we repeat all randomized approaches for 10 runs. To mitigate bias in the empirical study, two authors independently conducted the study and discussed all cases to reach consensus.

A major internal threat to validity comes from the approach implementation. We reduce this threat by conducting code reviews and outputting extensive logs. An author independently checked the correctness of code implementation. We also verified the soundness of our static analysis framework with over 50 carefully designed unit tests.

VII. Related Work

Understanding and Detecting Defects in DNNs. Discovering and mitigating the defects, faults, bugs, and failures in DNN based systems is an important research topic [58, 31, 14]. Following the taxonomy in [12], DNN defects are at four levels from bottom to top: (1) Platform-level defects: Defects can exist in real-world DL compilers and libraries. Approaches exist for understanding, detecting, and testing against these defects (e.g., [48, 36, 41, 51]). (2) Architecture-level defects: *Our work focuses on numerical defects, which is one type of architecture-level defects.* Automatic detection and localization techniques (e.g., [49, 18]) exist for other architecture-level defects such as suboptimal structure, activation function, initialization and shape mismatch [12]. (3) Model-level defects: Once a model is trained, its defects can be viewed as violations of desired properties which are discussed in detail in [53]. Some example defects are correctness [42, 10], robustness [47], and fairness [55] defects. (4) Interface-level defects: DNN-based systems, when deployed as services, expose interaction interfaces to users where defects may exist as empirical studies on real-world systems reveal [12, 45, 46].

Testing and Debugging for DNNs. A rich body of work exists for testing and debugging DNN defects [53]. Some representatives are DeepXplore [30] and DeepGauge [21]. Recent works enable automatic model debugging and repair via consistency checking [50], log checking [57], spectrum analysis [33], or analyzer-guided synthesis [40].

DNN Static Analysis. Another solution for eliminating DNN defects is conducting static analysis to rigorously guarantee the non-existence of defects [17, 2]. Though DNNs essentially conduct numerical computations, traditional numerical analysis tools [11, 38] are inefficient for DNN analysis due to lack of support for multi-dimensional tensor computations. Recently, static analysis tools customized for DNNs are emerging, which mainly focus on proposing tighter abstractions [7, 25, 28] or incorporating abstractions into training to improve robustness [16, 24, 60]. Besides robustness, static analysis has also be applied to rigorously bound model difference [29]. Our approach includes a static analysis framework customized for numerical defect detection and fixing.

Detecting and Exposing Numerical Defects in DNNs. Despite the widespread existence of numerical defect in real-world DNN-based systems [58, 12, 14], few automatic approaches exist for detecting and exposing these defects. To the best of our knowledge, DEBAR [59] and GRIST [52] are only two approaches. We discuss and compare with both approaches extensively in Sections III and V.

VIII. Conclusion

In this paper, we have presented a novel automatic approach RANUM for reliability assurance of DNNs against numerical defects. RANUM supports detection of potential numerical defects, confirmation of potential-defect feasibility, and suggestion of defect fixes. RANUM includes multiple novel extensions and optimizations upon existing tools, and includes three novel techniques. Our extensive evaluation on real-world DNN architectures has demonstrated high effectiveness and efficiency of RANUM compared to both the state-of-the-art approaches and developers' fixes.

Data Availability. All artifacts including the PDF supplementary materials (`suppl.pdf`), tool source code (`RANUM` folder), raw experiment data (`RANUM/results` folder), and logs of empirical study (`RANUM/empirical_study` folder) are available at `https://figshare.com/s/908f4981ee7e1c049c00`. The latest open source code of the tool is actively maintained at `https://github.com/llylly/RANUM`.

Acknowledgements

This work is sponsored by the National Key Research and Development Program of China under Grant No. 2019YFE0198100, the Innovation and Technology Commission of HKSAR under Grant No. MHP/055/19, and the National Natural Science Foundation of China under Grant No. 62161146003.

References

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

[2] A. Albarghouthi, "Introduction to neural network verification," *arXiv preprint arXiv:2109.10317*, 2021.

[3] Y. Bengio, N. Léonard, and A. Courville, "Estimating or propagating gradients through stochastic neurons for conditional computation," *arXiv preprint arXiv:1308.3432*, 2013.

[4] N. D. Bui, Y. Yu, and L. Jiang, "Infercode: Self-supervised learning of code representations by predicting subtrees," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1186–1197.

[5] P. Cousot and R. Cousot, "Static determination of dynamic properties of generalized type unions," *ACM SIGOPS Operating Systems Review*, vol. 11, no. 2, pp. 77–94, 1977.

[6] T. L. Foundation, "Onnx home," https://onnx.ai/, 2022, accessed: 2022-02-25.

[7] T. Gehr, M. Mirman, D. Drachsler-Cohen, P. Tsankov, S. Chaudhuri, and M. Vechev, "Ai2: Safety and robustness certification of neural networks with abstract interpretation," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 3–18.

[8] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *Proceedings of the fourteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings, 2011, pp. 315–323.

[9] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015.

[10] A. Guerriero, R. Pietrantuono, and S. Russo, "Operation is the hardest teacher: estimating dnn accuracy looking for mispredictions," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 348–358.

[11] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas, "The seahorn verification framework," in *International Conference on Computer Aided Verification*. Springer, 2015, pp. 343–361.

[12] N. Humbatova, G. Jahangirova, G. Bavota, V. Riccio, A. Stocco, and P. Tonella, "Taxonomy of real faults in deep learning systems," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1110–1121.

[13] N. Jay, N. Rotman, B. Godfrey, M. Schapira, and A. Tamar, "A deep reinforcement learning perspective on internet congestion control," in *International Conference on Machine Learning*. PMLR, 2019, pp. 3050–3059.

[14] E. Kloberdanz, K. G. Kloberdanz, and W. Le, "Deepstability: A study of unstable numerical methods and their solutions in deep learning," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 586–597. [Online]. Available: https://doi.org/10.1145/3510003.3510095

[15] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, 2012.

[16] L. Li, Z. Zhong, B. Li, and T. Xie, "Robustra: Training provable robust neural networks over reference adversarial space." in *IJCAI*, 2019, pp. 4711–4717.

[17] L. Li, T. Xie, and B. Li, "Sok: Certified robustness for deep neural networks," in *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, 22-26 May 2023*. IEEE, 2023.

[18] C. Liu, J. Lu, G. Li, T. Yuan, L. Li, F. Tan, J. Yang, L. You, and J. Xue, "Detecting tensorflow program bugs in real-world industrial environment," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 55–66.

[19] F. Liu, G. Li, B. Wei, X. Xia, Z. Fu, and Z. Jin, "A unified multi-task learning model for ast-level and token-level code completion," *Empirical Software Engineering*, vol. 27, no. 4, pp. 1–38, 2022.

[20] Z. Liu, P. Luo, X. Wang, and X. Tang, "Deep learning face attributes in the wild," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 3730–3738.

[21] L. Ma, F. Juefei-Xu, F. Zhang, J. Sun, M. Xue, B. Li, C. Chen, T. Su, L. Li, Y. Liu *et al.*, "Deepgauge: Multi-granularity testing criteria for deep learning systems," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 120–131.

[22] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards deep learning models resistant to adversarial attacks," in *International Conference on Learning Representations*, 2018.

[23] P. McCausland, "Self-driving uber car that hit and killed woman did not recognize that pedestrians jaywalk," 2022. [Online]. Available: https://www.nbcnews.com/tech/tech-news/self-driving-uber-car-hit-killed-woman-did-not-recognize-n1079281

[24] M. Mirman, T. Gehr, and M. Vechev, "Differentiable abstract interpretation for provably robust neural networks," in *International Conference on Machine Learning*. PMLR, 2018, pp. 3578–3586.

[25] M. N. Müller, G. Makarchuk, G. Singh, M. Püschel, and

M. Vechev, “Prima: general and precise neural network certification via scalable convex hull approximations,” *Proceedings of the ACM on Programming Languages*, vol. 6, no. POPL, pp. 1–33, 2022.

[26] A. Odena, C. Olsson, D. Andersen, and I. Goodfellow, “Tensorfuzz: Debugging neural networks with coverage-guided fuzzing,” in *International Conference on Machine Learning*. PMLR, 2019, pp. 4901–4911.

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

[28] B. Paulsen and C. Wang, “Linsyn: Synthesizing tight linear bounds for arbitrary neural network activation functions,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2022, pp. 357–376.

[29] B. Paulsen, J. Wang, J. Wang, and C. Wang, “Neurodiff: scalable differential verification of neural networks using fine-grained approximation,” in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 784–796.

[30] K. Pei, Y. Cao, J. Yang, and S. Jana, “Deepxplore: Automated whitebox testing of deep learning systems,” in *proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 1–18.

[31] H. V. Pham, S. Qian, J. Wang, T. Lutellier, J. Rosenthal, L. Tan, Y. Yu, and N. Nagappan, “Problems and opportunities in training deep learning software systems: An analysis of variance,” in *Proceedings of the 35th IEEE/ACM international conference on automated software engineering*, 2020, pp. 771–783.

[32] L. Powell, “The problem with artificial intelligence in security,” 2022. [Online]. Available: https://www.darkreading.com/threat-intelligence/the-problem-with-artificial-intelligence-in-security

[33] H. Qi, Z. Wang, Q. Guo, J. Chen, F. Juefei-Xu, L. Ma, and J. Zhao, “Archrepair: Block-level architecture-oriented repairing for deep neural networks,” *arXiv preprint arXiv:2111.13330*, 2021.

[34] T. Serra, C. Tjandraatmadja, and S. Ramalingam, “Bounding and counting linear regions of deep neural networks,” in *International Conference on Machine Learning*. PMLR, 2018, pp. 4558–4566.

[35] D. K. Sharma, S. K. Dhurandher, I. Woungang, R. K. Srivastava, A. Mohananey, and J. J. Rodrigues, “A machine learning-based protocol for efficient routing in opportunistic networks,” *IEEE Systems Journal*, vol. 12, no. 3, pp. 2207–2213, 2016.

[36] Q. Shen, H. Ma, J. Chen, Y. Tian, S. Cheung, and X. Chen, “A comprehensive study of deep learning compiler bugs,” in *ESEC/FSE ’21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, D. Spinellis, G. Gousios, M. Chechik, and M. D. Penta, Eds. ACM, 2021, pp. 968–980. [Online]. Available: https://doi.org/10.1145/3468264.3468591

[37] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton *et al.*, “Mastering the game of go without human knowledge,” *nature*, vol. 550, no. 7676, pp. 354–359, 2017.

[38] G. Singh, M. Püschel, and M. Vechev, “Fast polyhedra abstract domain,” in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, 2017, pp. 46–59.

[39] G. Singh, T. Gehr, M. Püschel, and M. Vechev, “An abstract domain for certifying neural networks,” *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–30, 2019.

[40] M. Sotoudeh and A. V. Thakur, “Provable repair of deep neural networks,” in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021, pp. 588–603.

[41] F. Tambon, A. Nikanjam, L. An, F. Khomh, and G. Antoniol, “Silent bugs in deep learning frameworks: An empirical study of keras and tensorflow,” *arXiv preprint arXiv:2112.13314*, 2021.

[42] S. Tizpaz-Niari, P. Čerǹy, and A. Trivedi, “Detecting and understanding real-world differential performance bugs in machine learning libraries,” in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 189–199.

[43] F. Tramer, N. Carlini, W. Brendel, and A. Madry, “On adaptive attacks to adversarial example defenses,” *Advances in Neural Information Processing Systems*, vol. 33, pp. 1633–1645, 2020.

[44] O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev *et al.*, “Grandmaster level in starcraft ii using multi-agent reinforcement learning,” *Nature*, vol. 575, no. 7782, pp. 350–354, 2019.

[45] C. Wan, S. Liu, H. Hoffmann, M. Maire, and S. Lu, “Are machine learning cloud apis used correctly?” in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 125–137.

[46] C. Wan, S. Liu, S. Xie, Y. Liu, H. Hoffmann, M. Maire, and S. Lu, “Automated testing of software that uses machine learning apis,” in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, 2022.

[47] J. Wang, J. Chen, Y. Sun, X. Ma, D. Wang, J. Sun, and P. Cheng, “Robot: robustness-oriented testing for deep learning systems,” in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 300–311.

[48] Z. Wang, M. Yan, J. Chen, S. Liu, and D. Zhang, “Deep learning library testing via effective model generation,” in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 788–799.

[49] M. Wardat, W. Le, and H. Rajan, “Deeplocalize: fault localization for deep neural networks,” in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 251–262.

[50] Y. Xiao, I. Beschastnikh, D. S. Rosenblum, C. Sun, S. Elbaum, Y. Lin, and J. S. Dong, “Self-checking deep neural networks in deployment,” in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 372–384.

[51] D. Xie, Y. Li, M. Kim, H. V. Pham, L. Tan, X. Zhang, and M. Godfrey, “Leveraging documentation to test deep learning library functions,” *arXiv preprint arXiv:2109.01002*, 2021.

[52] M. Yan, J. Chen, X. Zhang, L. Tan, G. Wang, and Z. Wang, “Exposing numerical bugs in deep learning via gradient back-propagation,” in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 627–638.

[53] J. M. Zhang, M. Harman, L. Ma, and Y. Liu, “Machine learning testing: Survey, landscapes and horizons,” *IEEE Transactions on Software Engineering*, 2020.

[54] K. Zhang, Z. Zhang, Z. Li, and Y. Qiao, “Joint face detection and alignment using multitask cascaded convolutional networks,” *IEEE signal processing letters*, vol. 23, no. 10, pp. 1499–1503, 2016.

[55] P. Zhang, J. Wang, J. Sun, G. Dong, X. Wang, X. Wang, J. S. Dong, and T. Dai, “White-box fairness testing through adversarial sampling,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 949–960.

[56] T. Zhang, C. Gao, L. Ma, M. Lyu, and M. Kim, “An empirical study of common challenges in developing deep learning

applications," in *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2019, pp. 104–115.

[57] X. Zhang, J. Zhai, S. Ma, and C. Shen, "Autotrainer: An automatic dnn training problem detection and repair system," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 359–371.

[58] Y. Zhang, Y. Chen, S.-C. Cheung, Y. Xiong, and L. Zhang, "An empirical study on tensorflow program bugs," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018, pp. 129–140.

[59] Y. Zhang, L. Ren, L. Chen, Y. Xiong, S.-C. Cheung, and T. Xie, "Detecting numerical bugs in neural network architectures," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 826–837.

[60] Y. Zhang, A. Albarghouthi, and L. D'Antoni, "Certified robustness to programmable transformations in lstms," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, M. Moens, X. Huang, L. Specia, and S. W. Yih, Eds. Association for Computational Linguistics, 2021, pp. 1068–1083.

[61] L. Zhu, Z. Liu, and S. Han, "Deep leakage from gradients," *Advances in neural information processing systems*, vol. 32, 2019.

SUPPLEMENTARY MATERIAL

A. *List of Supported Operators*

In this section, we provide a list of the 84 supported operators in the RANUM static analysis framework. These operators cover common operators used in DNNs. To the best of our knowledge, RANUM provides abstractions for the largest number of operator types among existing DNN abstraction frameworks. We believe the framework can be further extended to support other types of analysis, testing, and debugging for general DNN architectures or models such as robustness analysis and fairness testing.

In particular, when compared to the state-of-the-art tool DEBAR [59], the underlined 17 operators are newly supported by RANUM.

Sub	Add	Mul
Div	Pow	MatMul
Gemm	MaxPool	GlobalMaxPool
GlobalAveragePool	Cos	Sin
AveragePool	Conv	ConvTranspose
Pad	Reciprocal	Sqrt
Tanh	Relu	Softplus
LeakyRelu	Softsign	Sigmoid
Neg	Exp	Log
Softmax	LogSoftmax	Abs
Ceil	Floor	Sign
Reshape	Flatten	Transpose
Shape	Cast	Slice
Gather	GatherND	Squeeze
Unsqueeze	ScatterElements	Expand
Concat	Split	Identity
ConstantOfShape	RandomNormalLike	RandomUniformLike
RandomNormal	RandomUniform	Range
Constant	Less	LessOrEqual
Greater	GreaterOrEqual	Equal
Not	Or	Min
Max	Clip	Sum
ReduceMin	ReduceMax	ReduceSum
ReduceMean	ArgMax	ArgMin
Tile	NegativeLogLikelihoodLoss	Loop
SequenceInsert	BatchNormalization	OneHot
NonZero	Resize	ReduceProd
ReduceSumSquare	IsInf	IsNaN

B. *List of Operators with Potential Numerical Defects*

In this section, we provide a full list of DNN operators that may contain numerical defects, along with their invalid ranges $\mathcal{I}_{n_0,\mathsf{invalid}}$ (see definition of numerical defect in Definition 1) respectively. In the table, $U_{\min}$ and $U_{\max}$ stand for the minimum and maximum positive number of the input tensor's data type, respectively.

Op. Type	$\mathcal{I}_{n_0,\mathsf{invalid}}$
Pow	$[-U_{\min}, U_{\min}] \times (-\infty, -U_{\min}]$
Div	$\mathbb{R} \times [-U_{\min}, U_{\min}]$
Reciprocal	$[-U_{\min}, U_{\min}]$
Sqrt	$(-\infty, U_{\min}]$
Exp	$[\ln U_{\max}, \infty)$
Log	$(-\infty, U_{\min}]$
Range	$\mathbb{R} \times \mathbb{R} \times [-U_{\min}, U_{\min}]$
NegativeLogLikelihoodLoss	$[0, 0]$ for number of non-zero cells using mean reduction

C. *Detail Description of RANUM Static Analysis Framework*

In this section, we present the omitted details in Section III-A.

1) Abstraction Domain and Characteristics: We first formally define our abstraction domain: interval with tensor partitioning. Following the notation in abstract interpretation literature [5], suppose the tensor has m dimensions with shape $n_1 \times n_2 \times \cdots \times n_m$, we define the abstract domain as such:

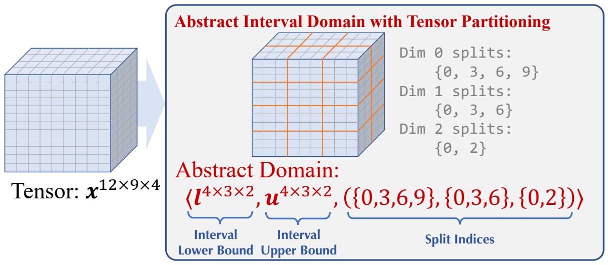
Abstract Interval Domain with Tensor Partitioning
Dim 0 splits:
{0, 3, 6, 9}
Dim 1 splits:
{0, 3, 6}
Dim 2 splits:
{0, 2}
Tensor: $\boldsymbol{x}^{12\times 9\times 4}$
Abstract Domain:
$\langle \boldsymbol{l}^{4\times 3\times 2}, \boldsymbol{u}^{4\times 3\times 2}, (\{0,3,6,9\},\{0,3,6\},\{0,2\})\rangle$
Interval Lower Bound
Interval Upper Bound
Split Indices

Fig. 5. Example of tensor partitioning. Tensor partitioning reduces the size of tensors in the abstract domain by sharing one interval bound among all elements inside one subblock.

$$\begin{aligned} \mathbb{A} := \{\langle \boldsymbol{l}, \boldsymbol{u}, (S_i)_{i=1}^m \rangle : \boldsymbol{l}, \boldsymbol{u} \in \mathbb{R}^{n'_1 \times \cdots \times n'_m} \\ |S_i| = n'_i, \forall s \in S_i, s \in \mathbb{N}_+, 0 \le s < n_i\}. \end{aligned} \tag{7}$$

Each element in $\mathbb{A}$ is a triplet $a = \langle \boldsymbol{l}, \boldsymbol{u}, (S_i)_{i=1}^m \rangle$, where the first two elements are subblock-wise interval lower bound and upper bound respectively, and the last element $(S_1, S_2, \ldots, S_m)$ contains m sets, where each set S_i corresponds to the *0-indexed* split indices for the i-th dimension to form subblocks. We use $S_i[j] \in \mathbb{N}_+$ to represent the j-th element of split index set S_i sorted in ascending order and define $S_i[|S_i|] = n_i$. Figure 5 illustrates this abstraction domain.

To define the correspondence between the abstract domain and concrete domain $\mathbb{C} := 2^{\mathbb{R}^{n_1 \times \cdots \times n_m}}$, we form Galois connection $\langle \mathbb{C}, \subseteq \rangle \underset{\gamma}{\overset{\alpha}{\rightleftharpoons}} \langle \mathbb{A}, \sqsubseteq \rangle$, where an abstraction function $\alpha : \mathbb{C} \to \mathbb{A}$ and the concretization function $\gamma : \mathbb{A} \to \mathbb{C}$ are defined as follows:

$$\begin{aligned} &\alpha(\mathcal{X}) = \langle \boldsymbol{l}, \boldsymbol{u}, (S_i)_{i=1}^m \rangle \quad \text{where} \\ &\quad \boldsymbol{l}_{i_1,i_2,\ldots,i_m} = \min_{\boldsymbol{x} \in \mathcal{X}} \min_{\substack{1 \le k \le m \\ S_k[i_k] \le j_k < S_k[i_k+1]}} \boldsymbol{x}_{j_1,j_2,\ldots,j_m}, \\ &\quad \boldsymbol{u}_{i_1,i_2,\ldots,i_m} = \max_{\boldsymbol{x} \in \mathcal{X}} \max_{\substack{1 \le k \le m \\ S_k[i_k] \le j_k < S_k[i_k+1]}} \boldsymbol{x}_{j_1,j_2,\ldots,j_m}, \end{aligned} \tag{8a}$$

$$\begin{aligned} \gamma(\langle \boldsymbol{l}, \boldsymbol{u}, (S_i)_{i=1}^m \rangle) = \{\boldsymbol{x} : \boldsymbol{l}_{i_1,i_2,\cdots,i_m} \le \boldsymbol{x}_{j_1,j_2,\cdots,j_m} \le \boldsymbol{u}_{i_1,i_2,\cdots,i_m}, \\ S_k[i_k] \le j_k < S_k[i_k+1], 1 \le k \le m\}. \end{aligned} \tag{8b}$$

In Equation (8a), the split indices $(S_i)_{i=1}^m$ can be arbitrarily chosen but need to be kept consistent with those in Equation (8b). Take Figure 5 as the example, to abstract a set of tensors $\mathcal{X}$ with shape $12 \times 9 \times 4$, we define split indices for each dimension respectively, then impose interval constraints on each subblock of the tensor. For example, $[\boldsymbol{l}_{0,0,0}, \boldsymbol{u}_{0,0,0}]$ constrain any element in $\boldsymbol{x}_{0:2,0:2,0:1}$, $[\boldsymbol{l}_{3,2,1}, \boldsymbol{u}_{3,2,1}]$ constrain any element in $\boldsymbol{x}_{9:11,6:8,2:3}$.

Abstraction Characteristics. There are multiple ways to compute the abstractions, and we design our particular computational algorithms to achieve soundness, tightness, and differentiability.

(1) **Soundness**: We guarantee that all abstractions are sound. Formally, suppose $a_{\boldsymbol{x}}$ and $a_{\boldsymbol{w}}$ are the abstraction for input and weights respectively, we guarantee $f_n^*(\gamma(a_{\boldsymbol{x}}); \gamma(a_{\boldsymbol{w}})) \subseteq \gamma(T_n^*(a_{\boldsymbol{x}}; a_{\boldsymbol{w}}))$ where $* \in \{\mathsf{in}, \mathsf{out}\}$, n is any node in the architecture, and $T_n^*(a_{\boldsymbol{x}}; a_{\boldsymbol{w}})$ is our computed abstract domain for

input or output of node n. The soundness property theoretically guarantees that our detection approach has no false negatives, i.e., flag all potential numerical defects, and the validness of generated preconditions (see Section II-B).

(2) **Tightness**: For most operators in DNN, given abstractions for its inputs, we compute the tightest possible interval abstraction for the output. Formally, for any atomic DNN operator *op* and any abstract domain $i \in \mathbb{A}$ of *op*'s input, if $op(\gamma(i)) \subseteq [\boldsymbol{l}, \boldsymbol{u}]$, i.e., $[\boldsymbol{l}, \boldsymbol{u}]$ is an interval abstraction of *op*'s output, then $\gamma(T_{op}(i)) \subseteq [\boldsymbol{l}, \boldsymbol{u}]$, i.e., our generated abstract domain $T_{op}(i)$ is always tighter or equal to any interval abstraction $[\boldsymbol{l}, \boldsymbol{u}]$. Such tightness can reduce false positives for detecting potential numerical defects and increase the span of generated failure-exhibiting intervals and fix intervals, which improves the quality of generated tests and preconditions.

(3) **Differentiability**: We compute differentiable abstractions. Concretely, if the input of node n_2 is deferentially dependent on output of node n_1, we can compute out gradients $\nabla_{\boldsymbol{l}_{n_1}/\boldsymbol{u}_{n_1}} \boldsymbol{l}_{n_2}/\boldsymbol{u}_{n_2}$, where $\langle \boldsymbol{l}_{n_1}, \boldsymbol{u}_{n_1}, (S_i^{n_1})_{i=1}^{\dim(n_1.\mathsf{out})} \rangle$ and $\langle \boldsymbol{l}_{n_2}, \boldsymbol{u}_{n_2}, (S_i^{n_2})_{i=1}^{\dim(n_2.\mathsf{in})} \rangle$ are abstractions of $f_{n_1}^{\mathsf{out}}(\cdot;\cdot)$ and $f_{n_2}^{\mathsf{in}}(\cdot;\cdot)$ respectively. When tightness and differentiability cannot be achieved at the same time (e.g., for `floor` operator, tight abstraction $\langle \boldsymbol{l}, \boldsymbol{u} \rangle \mapsto \langle \lfloor \boldsymbol{l} \rfloor, \lfloor \boldsymbol{u} \rfloor \rangle$ is not generally differentiable, and differentiable abstraction $\langle \boldsymbol{l}, \boldsymbol{u} \rangle \mapsto \langle \boldsymbol{l} - \mathbf{1}, \boldsymbol{u} \rangle$ is not tight), we implement two abstract algorithms to achieve tightness and differentiability respectively.

The existing approach DEBAR [59] also proposes a static analysis framework for DNN architectures with tensor partitioning. In contrast to our framework, the abstract domain in DEBAR contains affine equalities besides interval domains. Therefore, DEBAR can be tighter than ours in some cases and can produce fewer positives, but due to the additional complexity of affine equalities, DEBAR tends to use the coarest abstraction (i.e., tensor partitioning) granularity and supports fewer operators than ours. At the same time, our algorithms produce tightest interval abstractions while DEBAR has no tightness guarantee. As a result, RANUM detects more true numerical defects and has a comparable number of false positives (see Section V-A), and we can leverage the feasibility confirmation support in RANUM to filter out false positives.

*2) Initial Abstraction Construction with **Backward Fine-Grained Node Labelling**:* We construct abstract domains for initial nodes in two steps: First, we determine the tensor partitions, i.e., the split index sets $(S_i)_{i=1}^m$ (see Equation (7)), which decide the tightness and efficiency of our static analysis framework, because the tensor partitions of all other nodes will be solely dependent on the partitions of initial nodes as shown in Suppl. C3. Second, we compute the interval bounds $\boldsymbol{l}$ and $\boldsymbol{u}$.

We use the following principle to decide their tensor partitions: For nodes that are connected to operators requiring fine-grained abstractions (e.g., the `shape` input for operator `Reshape`) with valid paths which we will specify later, we construct tensor partitions with the finest granularity, i.e., $S_i = \{0, 1, \ldots, n_i - 1\}$. We call these nodes as fine-grained initial nodes, and starting from next paragraph we introduce our novel technique of *backward fine-grained node labelling* to find them out. For other nodes, we rely on downstream task requirements and user specifications to determine the partitions. For example, for fix generation (see Section III-C), we use the coarsest granularity by default.

Backward Fine-Grained Node Labelling. The fine-grained initial nodes are those starting a valid path in DNN computational graph $\mathcal{G}$, where a path is valid if and only if the path does not traverse through fine-grained stopping operators and terminates at a fine-grained requiring operator with some specific input index. Fine-grained requiring operators are those taking indices as inputs (so each index value is important and finest partition granularity is needed) and those controlling the loop execution (so we can need finest granularity to know looping times to unfold). Fine-grained stopping operators are those which output is independent of input abstraction granularity (so granularity of preceding nodes do not matter). Detail lists of fine-grained stopping operators and fine-grained requiring operators are provided in Suppl. C4.

To find out fine-grained initial nodes, we propose backward fine-grained node labelling, which is similar to data dependency analysis in traditional programs: First, we invert all edges of the given computational graph $\mathcal{G} = \langle \mathcal{V}, \mathcal{E} \rangle$ to get $\mathcal{G}' = \langle \mathcal{V}, \mathcal{E}' \rangle$. Second, we attach a boolean label for each node $n \in \mathcal{V}$ and initialize them with `False`. Third, we do topology sort on $\mathcal{G}'$. When encountering a node n with `True`, if the node does not contain a fine-grained stopping operator, we propagate this label to all its subsequent nodes; otherwise, we propagate `False`. When encountering a node n with `False`, if the node is a fine-grained requiring operator, we propagate `True` to corresponding subsequent nodes and `False` to others; otherwise propagate `False` to all subsequent nodes. Fourth, we collect all labels for initial nodes. The nodes with `True` label are fine-grained initial nodes.

To this point, we have determined the tensor partitions $(S_i)_{i=1}^m$ for all initial nodes, and we now compute $\boldsymbol{l}$ and $\boldsymbol{u}$ and thus finish the abstraction domain construction for the valid ranges of initial nodes. Initial nodes are further divided into input, weight, and constant nodes. In practice, most of weight nodes and all constant nodes have their initial values stored in the ONNX file, and we directly use Equation (8a) with these initial values to compute $\boldsymbol{l}$ and $\boldsymbol{u}$. Otherwise, we rely on user specifications and built-in heuristics to determine $\boldsymbol{l}$ and $\boldsymbol{u}$.

*3) Internal Abstraction with **Dynamic Partitioning**:* For all supported operator types (listed in Suppl. A), we propose concrete algorithms to compute abstract domains for output with dynamic tensor partitioning. Formally, for each operator *op*, we construct the computable function $T_{op} : \mathbb{A} \to \mathbb{A}$ that satisfies soundness and tentatively satisfies tightness and differentiability, where $\mathbb{A}$ is the abstract domain defined in Equation (7). Therefore, following the computational procedure introduced before, we can compute end-to-end abstractions for all nodes in the given DNN architecture. Given the limited space, we only describe the algorithms for these representative operators: `MatMul`, `Conv`, and `Softmax`.

`MatMul`. The `MatMul` operator computes the matrix multiplication of two operands. This operator is widely used in DNNs to express fully-connected layers. To simplify the narration, we focus on two-dimensional case where we compute $op(\boldsymbol{A}, \boldsymbol{B}) = \boldsymbol{C} = \boldsymbol{AB}$ with $\boldsymbol{A} \in \mathbb{R}^{n\times m}$ and $\boldsymbol{B} \in \mathbb{R}^{m\times l}$, and extensions to other dimensions by abstracting the broadcasting mechanism can be found in our open-source implementation.

We denote the input abstractions of op by $a = \langle \boldsymbol{L}_a, \boldsymbol{U}_a, (S_A^1, S_A^2)\rangle$ and $b = \langle \boldsymbol{L}_b, \boldsymbol{U}_b, (S_B^1, S_B^2)\rangle$ respectively. First, we compute the union $U = S_A^2 \cup S_B^1$. Second, we dynamically partition both a and b with split points (S_A^1, U) and (U, S_B^2) respectively and get $a' = \langle \boldsymbol{L}'_a, \boldsymbol{U}'_a, (S_A, U)\rangle$ and $b' = \langle \boldsymbol{L}'_b, \boldsymbol{U}'_b, (U, S_B^2)\rangle$. Note that a and a' (or b and b') correspond to the same concrete domain, but a' and b' have finer or equal partition granularity than a and b. Third, we compute output abstraction $T_{op}(a, b) = c = \langle \boldsymbol{L}_c, \boldsymbol{U}_c, (S_A^1, S_B^2)\rangle$

$$\begin{aligned} \text{where} \quad (\boldsymbol{L}_c)_{ij} &= \sum_{k=1}^{|U|} \boldsymbol{v}_k \min_{\boldsymbol{A}\in\{\boldsymbol{L}'_a, \boldsymbol{U}'_a\}, \boldsymbol{B}\in\{\boldsymbol{L}'_b, \boldsymbol{U}'_b\}} \boldsymbol{A}_{ik}\boldsymbol{B}_{kj}, \\ (\boldsymbol{U}_c)_{ij} &= \sum_{k=1}^{|U|} \boldsymbol{v}_k \max_{\boldsymbol{A}\in\{\boldsymbol{L}'_a, \boldsymbol{U}'_a\}, \boldsymbol{B}\in\{\boldsymbol{L}'_b, \boldsymbol{U}'_b\}} \boldsymbol{A}_{ik}\boldsymbol{B}_{kj}, \\ \boldsymbol{v}_k &= U[k] - U[k-1]. \end{aligned} \tag{9}$$

This formulation can guarantee the tightness but is not efficient for tensor computation due to inner minimum and maximum. Therefore, we also implement a fast-mode abstraction which trade-off tightness for efficiency: $T'_{op}(a, b) = c' = \langle \boldsymbol{L}'_c, \boldsymbol{U}'_c, (S_A^1, S_B^2)\rangle$ where

$$\begin{aligned} \boldsymbol{L}_c^{'} =& \boldsymbol{U}_a^{'-}\boldsymbol{v}\boldsymbol{U}_b^{'-} + \boldsymbol{U}_a^{'0}\boldsymbol{v}\boldsymbol{L}_b^{'-} + \boldsymbol{U}_a^{'+}\boldsymbol{v}\boldsymbol{L}_b^{'-} + \\ & \boldsymbol{L}_a^{'-}\boldsymbol{v}\boldsymbol{U}_b^{'0} + \boldsymbol{U}_a^{'0}\boldsymbol{v}\boldsymbol{L}_b^{'0} + \boldsymbol{L}_a^{'0}\boldsymbol{v}\boldsymbol{U}_b^{'0} + \boldsymbol{U}_a^{'+}\boldsymbol{v}\boldsymbol{L}_b^{'0} + \\ & \boldsymbol{L}_a^{'-}\boldsymbol{v}\boldsymbol{U}_b^{'+} + \boldsymbol{L}_a^{'0}\boldsymbol{v}\boldsymbol{U}_b^{'+} + \boldsymbol{L}_a^{'+}\boldsymbol{v}\boldsymbol{L}_b^{'+}, \\ \boldsymbol{U}_c^{'} =& \boldsymbol{L}_a^{'-}\boldsymbol{v}\boldsymbol{L}_b^{'-} + \boldsymbol{L}_a^{'0}\boldsymbol{v}\boldsymbol{L}_b^{'-} + \boldsymbol{L}_a^{'+}\boldsymbol{v}\boldsymbol{U}_b^{'-} + \\ & \boldsymbol{L}_a^{'-}\boldsymbol{v}\boldsymbol{L}_b^{'0} + \boldsymbol{L}_a^{'0}\boldsymbol{v}\boldsymbol{L}_b^{'0} + \boldsymbol{U}_a^{'0}\boldsymbol{v}\boldsymbol{U}_b^{'0} + \boldsymbol{U}_a^{'+}\boldsymbol{v}\boldsymbol{U}_b^{'0} + \\ & \boldsymbol{U}_a^{'-}\boldsymbol{v}\boldsymbol{L}_b^{'+} + \boldsymbol{U}_a^{'0}\boldsymbol{v}\boldsymbol{U}_b^{'+} + \boldsymbol{U}_a^{'+}\boldsymbol{v}\boldsymbol{U}_b^{'+}. \end{aligned} \tag{10}$$

In the above equation, for any $* \in \{a, b\}$, let $\circ$ be the elementwise (Hadamard) product,

$$\begin{aligned} \boldsymbol{L}_*^{'-} =& \boldsymbol{L}_*^{'} \circ \mathbb{I}[\boldsymbol{U}_*^{'} < 0], \boldsymbol{U}_*^{'-} = \boldsymbol{U}_*^{'} \circ \mathbb{I}[\boldsymbol{U}_*^{'} < 0], \\ \boldsymbol{L}_*^{'0} =& \boldsymbol{L}_*^{'} \circ \mathbb{I}[\boldsymbol{L}_*^{'} \le 0, \boldsymbol{U}_*^{'} \ge 0], \boldsymbol{U}_*^{'0} = \boldsymbol{U}_*^{'} \circ \mathbb{I}[\boldsymbol{L}_*^{'} \le 0, \boldsymbol{U}_*^{'} \ge 0], \\ \boldsymbol{L}_*^{'+} =& \boldsymbol{L}_*^{'} \circ \mathbb{I}[\boldsymbol{L}_*^{'} > 0], \boldsymbol{U}_*^{'+} = \boldsymbol{U}_*^{'} \circ \mathbb{I}[\boldsymbol{L}_*^{'} > 0]. \end{aligned} \tag{11}$$

From Equation (10), we can observe that the abstraction can be easily implemented with tensor computations. In Suppl. C5, we prove the soundness and tightness of these abstractions.

`Conv`. The `Conv` operator computes the discrete convolution of two operands. This operator is widely used in convolutional neural networks (CNNs, [15]). To simplify the narration, we focus on single-channel single-stride two-dimensional case where we compute $op(\boldsymbol{A}, \boldsymbol{W}) = \boldsymbol{C}$ with $\boldsymbol{A}$ being the input matrix and $\boldsymbol{W}$ being the convolution kernel. Extensions to general cases are provided in our open-source code.

We first dynamically split the kernel abstraction to the finest granularity. Second, we compute the receptive field of each output position, which is the sub-region of $\boldsymbol{A}$ that decides each output position. Third, we inspect the alignment between receptive fields and $\boldsymbol{A}$'s partitions. If neighboring output positions have their receptive fields partitioned in the same sub-block of $\boldsymbol{A}$, it means that these positions can be abstracted by a single interval, i.e., these positions can be partitioned together. Fourth, using this principle, we derive the paritions of the output tensor, and repeat the input tensors accordingly so that the abstract computation can be written as convolutional operations. Last, we modify the abstraction computation equations in Equation (10) by replacing matrix multiplication with convolution to compute the abstraction of output $\boldsymbol{C}$.

`Softmax`. The `Softmax` operator computes normalized exponential values for the given input. `Softmax` is widely deployed for classification tasks to yield the normalized confidence. In one-dimensional case, for input $\boldsymbol{x} \in \mathbb{R}^n$, a `Softmax` operator outputs $op(\boldsymbol{x}) = \frac{\exp(\boldsymbol{x})}{\sum_{i=1}^n \exp(\boldsymbol{x})_i}$. The output abstraction of `Softmax` operator op can be thus computed: $T_{op}(\langle \boldsymbol{l}, \boldsymbol{u}, (S)\rangle) = \langle \boldsymbol{l}^o, \boldsymbol{u}^o, (S)\rangle$ where

$$\begin{aligned} &\boldsymbol{l}_i^o = \frac{\exp(\boldsymbol{l}_i)}{\exp(\boldsymbol{u})^\intercal \boldsymbol{v} - \exp(\boldsymbol{u}_i) + \exp(\boldsymbol{l}_i)}, \boldsymbol{u}_i^o = \frac{\exp(\boldsymbol{u}_i)}{\exp(\boldsymbol{l})^\intercal \boldsymbol{v} - \exp(\boldsymbol{l}_i) + \exp(\boldsymbol{u}_i)}, \\ &\boldsymbol{v}_k = S[k] - S[k-1]. \end{aligned} \tag{12}$$

The output abstraction's partition is dynamically decided by the input abstraction's partition. We prove the soundness and tightness of the abstraction in Suppl. C5.

Remark. As we can see, the novel dynamic partition technique is incorporated in the computation process of each operator's abstraction. The soundness and tightness of our designed abstractions are immediately achieved by design, and the differentiability of our designed abstractions relies on the auto-differentiation functionality of popular DL libraries like `PyTorch` [27] and `Tensorflow` [1] where we use `PyTorch` for implementation. The abstractions for DNNs are also implemented for other applications, e.g., for robustness verification [39, 24]. However, our abstractions are tailored for tensor partitioned interval domain which is particularly suitable for testing and debugging for numerical failures as discussed before. To the best of our knowledge, these abstractions are the first that achieve soundness, tightness, and differentiability.

4) List of Fine-Grained Requiring and Stopping Operators: We use fine-grained requiring and fine-grained stopping operators in Suppl. C2. Among all supported operators, the fine-grained requiring operators are

> `Reshape` (input #2), `Slice` (input #2, #3, #4, #5), `Squeeze` (input #2), `Unsqueeze` (input #2), `Tile` (input #2, #3), `Loop` (input #1, #2), `SequenceInsert` (input #3), `ConstantOfShape` (input #1), `Gather` (input #2), `GatherND` (input #2), `ReduceSum` (input #2), `ScatterElements` (input #2), `Expand` (input #2), `Split` (input #2), `Pad` (input #2, #3), `NegativeLogLikelihoodLoss` (input #2), `Clip` (input #2, #3), `OneHot` (input #2), `Resize` (input #2, #3, #4).

The fine-grained stopping operators are `Shape`, `RandomNormalLike`, and `RandomUniformLike`.

5) Proofs: Here we present the omitted proofs in Suppl. C3.

`MatMul`.

Theorem 1 (tightness of first abstraction)**.** *Suppose op is the* `MatMul` *operator and T_{op} is as defined in Equation* (9)*, then if $op(\gamma(a), \gamma(b)) \subseteq [\boldsymbol{L}, \boldsymbol{U}]$ for $a, b \in \mathbb{A}$, $\gamma(T_{op}(a, b)) \subseteq [\boldsymbol{L}, \boldsymbol{U}]$.*

Proof. We use $\boldsymbol{L}_a$ and $\boldsymbol{U}_a$ to denote the element-wise interval lower and upper bounds of $\gamma(a)$; and use $\boldsymbol{L}_b$ and $\boldsymbol{U}_b$ to denote these bounds of $\gamma(b)$ respectively, where a formal definition is in Equation (8b). Then, there exist $\boldsymbol{A}$ and $\boldsymbol{B}$ with $\boldsymbol{L}_a \le \boldsymbol{A} \le \boldsymbol{U}_a$ and $\boldsymbol{L}_b \le \boldsymbol{B} \le \boldsymbol{U}_b$, such that

$$(\boldsymbol{AB})_{ij} = \sum_{k=1}^{l} \min\{\boldsymbol{L}_{a,ik}\boldsymbol{L}_{b,kj}, \boldsymbol{L}_{a,ik}\boldsymbol{U}_{b,kj}, \boldsymbol{U}_{a,ik}\boldsymbol{L}_{b,kj}, \boldsymbol{U}_{a,ik}\boldsymbol{U}_{b,kj}\}, \tag{13}$$

$$\text{or } (\boldsymbol{AB})_{ij} = \sum_{k=1}^{l} \max\{\boldsymbol{L}_{a,ik}\boldsymbol{L}_{b,kj}, \boldsymbol{L}_{a,ik}\boldsymbol{U}_{b,kj}, \boldsymbol{U}_{a,ik}\boldsymbol{L}_{b,kj}, \boldsymbol{U}_{a,ik}\boldsymbol{U}_{b,kj}\}. \tag{14}$$

By definition, we have

$$\boldsymbol{L}_{ij} \le \text{Equation (13)}, \boldsymbol{U}_{ij} \ge \text{Equation (14)}. \tag{15}$$

We let $\boldsymbol{L}_c'$ and $\boldsymbol{U}_c'$ to denote the element-wise interval lower and upper bounds for $\gamma(T_{op}(a, b))$, let $S_A^1[i'] \le i-1 < S_A^1[i'+1]$ and $S_B^2[j'] \le j-1 < S_B^2[j'+1]$, then from Equation (9),

$$\begin{aligned}
&(\boldsymbol{L}_c')_{ij} = (\boldsymbol{L}_c)_{i'j'} \\
=&\sum_{k=1}^{|U|} \sum_{k'=U[k-1]+1}^{U[k]} \min\{\boldsymbol{L}_{a,ik'}\boldsymbol{L}_{b,k'j}, \boldsymbol{L}_{a,ik'}\boldsymbol{U}_{b,k'j}, \\
&\qquad \boldsymbol{U}_{a,ik'}\boldsymbol{L}_{b,k'j}, \boldsymbol{U}_{a,ik'}\boldsymbol{U}_{b,k'j}\} \qquad (16) \\
=&\sum_{k=1}^{l} \min\{\boldsymbol{L}_{a,ik}\boldsymbol{L}_{b,kj}, \boldsymbol{L}_{a,ik}\boldsymbol{U}_{b,kj}, \boldsymbol{U}_{a,ik}\boldsymbol{L}_{b,kj}, \boldsymbol{U}_{a,ik}\boldsymbol{U}_{b,kj}\} \\
=&\text{Equation (13)} \ge \boldsymbol{L}_{ij}.
\end{aligned}$$

Similarly, $(\boldsymbol{U}_c')_{ij} \le \boldsymbol{U}_{ij}$. Thus, $\gamma(T_{op}(a, b)) \subseteq [\boldsymbol{L}, \boldsymbol{U}]$. □

Theorem 2 (soundness of first abstraction)**.** *Suppose op is the* `MatMul` *opeartor and T_{op} is as defined in Equation* (9)*, then $op(\gamma(a), \gamma(b)) \subseteq \gamma(T_{op}(a, b))$.*

Proof. We use $\boldsymbol{L}_a$ and $\boldsymbol{U}_a$ to denote the element-wise interval lower and upper bounds of $\gamma(a)$; and use $\boldsymbol{L}_b$ and $\boldsymbol{U}_b$ to denote these bounds of $\gamma(b)$ respectively, where a formal definition is in Equation (8b). For any $\boldsymbol{A}$ and $\boldsymbol{B}$ such that $\boldsymbol{L}_a \le \boldsymbol{A} \le \boldsymbol{U}_a$ and $\boldsymbol{L}_b \le \boldsymbol{B} \le \boldsymbol{U}_b$,

$$\begin{aligned}
(\boldsymbol{AB})_{ij} &= \sum_{k=1}^{l} \boldsymbol{A}_{ik}\boldsymbol{B}_{kj} \\
&\ge \min\{\boldsymbol{L}_{a,ik}\boldsymbol{L}_{b,kj}, \boldsymbol{L}_{a,ik}\boldsymbol{U}_{b,kj}, \boldsymbol{U}_{a,ik}\boldsymbol{L}_{b,kj}, \boldsymbol{U}_{a,ik}\boldsymbol{U}_{b,kj}\} \\
&=: (\boldsymbol{L}_{ab}')_{ij}
\end{aligned} \tag{17}$$

and

$$\begin{aligned}
(\boldsymbol{AB})_{ij} &\le \max\{\boldsymbol{L}_{a,ik}\boldsymbol{L}_{b,kj}, \boldsymbol{L}_{a,ik}\boldsymbol{U}_{b,kj}, \boldsymbol{U}_{a,ik}\boldsymbol{L}_{b,kj}, \boldsymbol{U}_{a,ik}\boldsymbol{U}_{b,kj}\} \\
&=: (\boldsymbol{U}_{ab}')_{ij}.
\end{aligned} \tag{18}$$

Thus, $op(\gamma(a), \gamma(b)) \subseteq [\boldsymbol{L}_{ab}', \boldsymbol{U}_{ab}']$. On the other hand, $\gamma(T_{op}(a, b)) = [\boldsymbol{L}_{ab}', \boldsymbol{U}_{ab}']$ as seen from Equation (16). Therefore, $op(\gamma(a), \gamma(b)) \subseteq \gamma(T_{op}(a, b))$. □

Theorem 3 (soundness of second abstraction)**.** *Suppose op is the* `MatMul` *operator and T_{op}' is as defined in Equations* (10) *and* (11)*, then $op(\gamma(a), \gamma(b)) \subseteq \gamma(T_{op}'(a, b))$.*

Proof. We use $\boldsymbol{L}_a$ and $\boldsymbol{U}_a$ to denote the element-wise interval lower and upper bounds of $\gamma(a)$; and use $\boldsymbol{L}_b$ and $\boldsymbol{U}_b$ to denote these bounds of $\gamma(b)$ respectively, where a formal definition is in Equation (8b). We define $\boldsymbol{L}_{ab}'$ and $\boldsymbol{U}_{ab}'$ by Equations (17) and (18). From the proof of Theorem 2, we have $op(\gamma(a), \gamma(b)) \subseteq [\boldsymbol{L}_{ab}', \boldsymbol{U}_{ab}']$. We now only need to show that $[\boldsymbol{L}_{ab}', \boldsymbol{U}_{ab}'] \subseteq \gamma(T_{op}'(a, b))$.

$\gamma(T_{op}'(a, b))$ imposes independent interval abstractions element-wise, therefore, we study each element independently. For the element (i, j), from Equations (10) and (11), the interval lower bound of $\gamma(T_{op}'(a, b))$, namely $(\boldsymbol{L}_c)_{ij}$, satisfies

$$\begin{aligned}
&(\boldsymbol{L}_c)_{ij} = \sum_{k=1}^{l} (\boldsymbol{U}_a^-)_{ik}(\boldsymbol{U}_b^-)_{kj} + (\boldsymbol{U}_a^0)_{ik}(\boldsymbol{L}_b^-)_{kj} + (\boldsymbol{U}_a^+)_{ik}(\boldsymbol{L}_b^-)_{kj} + \\
&(\boldsymbol{L}_a^-)_{ik}(\boldsymbol{U}_b^0)_{kj} + (\boldsymbol{U}_a^0)_{ik}(\boldsymbol{L}_b^0)_{kj} + (\boldsymbol{L}_a^0)_{ik}(\boldsymbol{U}_b^0)_{kj} + (\boldsymbol{U}_a^+)_{ik}(\boldsymbol{L}_b^0)_{kj} + \\
&(\boldsymbol{L}_a^-)_{ik}(\boldsymbol{U}_b^+)_{kj} + (\boldsymbol{L}_a^0)_{ik}(\boldsymbol{U}_b^+)_{kj} + (\boldsymbol{L}_a^+)_{ik}(\boldsymbol{L}_b^+)_{kj}.
\end{aligned} \tag{19}$$

By Equation (11),

- when $\boldsymbol{L}_{a,ik} \le \boldsymbol{U}_{a,ik} < 0$,
  $(\boldsymbol{L}_a^-)_{ik} = \boldsymbol{L}_{a,ik}, (\boldsymbol{U}_a^-)_{ik} = \boldsymbol{U}_{a,ik}, (\boldsymbol{L}_a^0)_{ik} = 0,$
  $(\boldsymbol{U}_a^0)_{ik} = 0, (\boldsymbol{L}_a^+)_{ik} = 0, (\boldsymbol{U}_a^+)_{ik} = 0;$
- when $\boldsymbol{L}_{a,ik} \le 0 \le \boldsymbol{U}_{a,ik}$,
  $(\boldsymbol{L}_a^-)_{ik} = 0, (\boldsymbol{U}_a^-)_{ik} = 0, (\boldsymbol{L}_a^0)_{ik} = \boldsymbol{L}_{a,ik},$
  $(\boldsymbol{U}_a^0)_{ik} = \boldsymbol{U}_{a,ik}, (\boldsymbol{L}_a^+)_{ik} = 0, (\boldsymbol{U}_a^+)_{ik} = 0;$
- when $0 < \boldsymbol{L}_{a,ik} \le \boldsymbol{U}_{a,ik}$,
  $(\boldsymbol{L}_a^-)_{ik} = 0, (\boldsymbol{U}_a^-)_{ik} = 0, (\boldsymbol{L}_a^0)_{ik} = 0,$
  $(\boldsymbol{U}_a^0)_{ik} = 0, (\boldsymbol{L}_a^+)_{ik} = \boldsymbol{L}_{a,ik}, (\boldsymbol{U}_a^+)_{ik} = \boldsymbol{U}_{a,ik}.$

Similarly for $(\boldsymbol{L}_b^-)_{kj}$, $(\boldsymbol{U}_b^-)_{kj}$, $(\boldsymbol{L}_b^0)_{kj}$, $(\boldsymbol{U}_b^0)_{kj}$, $(\boldsymbol{L}_b^+)_{kj}$ and $(\boldsymbol{U}_b^+)_{kj}$. Thus, by enumerating all cases, we have

$$\begin{aligned}
\text{Equation (19)} \le \min\{&\boldsymbol{L}_{a,ik}\boldsymbol{L}_{b,kj}, \boldsymbol{L}_{a,ik}\boldsymbol{U}_{b,kj}, \\
&\boldsymbol{U}_{a,ik}\boldsymbol{L}_{b,kj}, \boldsymbol{U}_{a,ik}\boldsymbol{U}_{b,kj}\} = (\boldsymbol{L}_{ab}')_{ij}.
\end{aligned} \tag{20}$$

Similarly, the interval lower bound of $\gamma(T_{op}'(a, b))$, namely $(\boldsymbol{U}_c)_{ij}$,

$$\begin{aligned}
(\boldsymbol{U}_c)_{ij} \ge \max\{&\boldsymbol{L}_{a,ik}\boldsymbol{L}_{b,kj}, \boldsymbol{L}_{a,ik}\boldsymbol{U}_{b,kj}, \\
&\boldsymbol{U}_{a,ik}\boldsymbol{L}_{b,kj}, \boldsymbol{U}_{a,ik}\boldsymbol{U}_{b,kj}\} = (\boldsymbol{U}_{ab}')_{ij}.
\end{aligned} \tag{21}$$

Thus, $[\boldsymbol{L}_{ab}', \boldsymbol{U}_{ab}'] \subseteq \gamma(T_{op}'(a, b))$. □

`Softmax`.

Theorem 4 (tightness)**.** *Suppose $op : \mathbb{R}^n \to \mathbb{R}^n$ is the* `Softmax` *operator and T_{op} is as defined in Equation* (12)*, then if $op(\gamma(\langle \boldsymbol{l}, \boldsymbol{u}, (S) \rangle)) \subseteq [\boldsymbol{l}^r, \boldsymbol{u}^r]$, then $\gamma(T_{op}(\langle \boldsymbol{l}, \boldsymbol{u}, (S) \rangle)) \subseteq [\boldsymbol{l}^r, \boldsymbol{u}^r]$.*

Proof. We use $\boldsymbol{l}'$ and $\boldsymbol{u}'$ to denote the element-wise interval lower and upper bounds of $\gamma(\langle \boldsymbol{l}, \boldsymbol{u}, (S) \rangle)$. Formally, $\boldsymbol{l}_i' = \boldsymbol{l}_{i'}$

and $\boldsymbol{u}'_i = \boldsymbol{u}_{i'}$ where $S[i'] \leq i - 1 < S[i'+1]$. Then, for each i, by setting $\boldsymbol{x}_i = \boldsymbol{l}_i$ and $\boldsymbol{x}_j = \boldsymbol{u}_j$ for all $j \neq i$, we get

$$\begin{aligned} \boldsymbol{l}^r_i &\leq \frac{\exp(\boldsymbol{l}'_i)}{\exp(\boldsymbol{l}'_i) + \sum_{k=1,k\neq i}^n \exp(\boldsymbol{u}'_k)} \\ &= \frac{\exp(\boldsymbol{l}_{i'})}{\exp(\boldsymbol{l}_{i'}) + \sum_{k=1}^n \exp(\boldsymbol{u}'_k) - \exp(\boldsymbol{u}'_i)} \\ &= \frac{\exp(\boldsymbol{l}_{i'})}{\exp(\boldsymbol{l}_{i'}) + \boldsymbol{v}^\intercal \exp(\boldsymbol{u}_k) - \exp(\boldsymbol{u}_{i'})} = \boldsymbol{l}^o_{i'}. \end{aligned} \tag{22}$$

In addition, by setting $\boldsymbol{x}_i = \boldsymbol{u}_i$ and $\boldsymbol{x}_j = \boldsymbol{l}_j$ for all $j \neq i$, we get $\boldsymbol{u}^r_i \geq \boldsymbol{u}^o_{i'}$. Here, $\boldsymbol{l}^o$ and $\boldsymbol{u}^o$ are as defined in Equation (12). Combining these two arguments, we get

$$\gamma(T_{op}(\langle \boldsymbol{l}, \boldsymbol{u}, (S)\rangle)) \subseteq [\boldsymbol{l}^r, \boldsymbol{u}^r].$$

□

Theorem 5 (soundness)**.** *Suppose op* : $\mathbb{R}^n \to \mathbb{R}^n$ *is the* `Softmax` *operator and* T_{op} *is as defined in Equation* (12)*, then* $op(\gamma(\langle \boldsymbol{l}, \boldsymbol{u}, (S)\rangle)) \subseteq \gamma(T_{op}(\langle \boldsymbol{l}, \boldsymbol{u}, (S)\rangle))$.

Proof. Leveraging the fact that softmax function is monotinically increasing, i.e., $\frac{\mathrm{d}op(\boldsymbol{x})_i}{\mathrm{d}\boldsymbol{x}_j} > 0$, we have $\gamma(\langle \boldsymbol{l}, \boldsymbol{u}, (S)\rangle)_i \in [\boldsymbol{l}^o_{i'}, \boldsymbol{u}^o_{i'}]$, where $S[i'] \leq i - 1 < S[i'+1]$. Since $[\boldsymbol{l}^o_{i'}, \boldsymbol{u}^o_{i'}] = \gamma(T_{op}(\langle \boldsymbol{l}, \boldsymbol{u}, (S)\rangle))_i$ by our definition of T_{op}, $op(\gamma(\langle \boldsymbol{l}, \boldsymbol{u}, (S)\rangle)) \subseteq \gamma(T_{op}(\langle \boldsymbol{l}, \boldsymbol{u}, (S)\rangle))$ follows. □

D. Hyperparameters

In this section, we listed hyperparameters in our two-step system test generation technique.

The system test generation approach in RANUM is introduced in Section III-B. The approach is composed of two steps: generation of unit test $\langle \boldsymbol{w}_{\mathsf{infer}}, \boldsymbol{x}_{\mathsf{infer}}\rangle$, and generation of training example $\boldsymbol{x}_{\mathsf{train}}$. For the unit test generation, we use the Adam optimizer with learning rate 1 and maximum iteration 100. For the training test generation, we target for training example under learning rate $\gamma = 1$ and the approach has similar performance under other learning rates. We follow the convention in DLG attack, where we use the L-BFGS method as the optimizer for gradient-based minimization. We terminate the method and return "`failed`" if either the running time exceeds 1800 s (universal execution time limit for all experimental evaluations), or a failure-exhibiting training is not found after 300 iterations of L-BFGS optimization.

E. Ablation Study of System Test Generation

RANUM contains the novel technique of two-step generation for generating failure-exhibiting system tests: it first generates failure-exhibiting unit tests with gradient back-propagation, then generates failure-exhibiting training example via the extended DLG attack. To isolate the impact of RANUM at each step, we replace either step with random sampling: "Random + RANUM" which first generates failure-exhibiting unit tests via random sampling, then generates training example via RANUM at the second task; "RANUM + Random" which first generates failure-exhibiting unit tests via RANUM, then generates training example via random sampling. We follow the same evaluation protocol as in Section V-B in this ablation study. We find that "RANUM + Random" takes 9.38X running time than RANUM and fails for 68 runs (RANUM only fails for 57 runs); and "Random + RANUM" fails for 113 runs (roughly 2X fail runs compared to RANUM). This study implies that, RANUM's technique helps to improve effectiveness and efficiency at both steps of failure-exhibiting system test generation compared to the pure random baseline. The improvement for the first step is mainly from the effectiveness perspective, and the improvement for the second step is mainly from the efficiency perspective.