

ARGUS: Effectively Detecting Tensor Shape Errors in Deep Learning Networks via Concolic Execution

Zihao Luo Baojian Hua*

School of Software Engineering, University of Science and Technology of China
Suzhou Institute for Advanced Research, University of Science and Technology of China
lzhcsu@mail.ustc.edu.cn bjhua@ustc.edu.cn

Abstract—As deep learning programs are increasingly deployed in safety-critical domains, the correctness and reliability of these programs have become a growing concern. Unfortunately, recent studies have shown that current deep learning programs still suffer from tensor-shape errors, undermining their security guarantees. Existing approaches for detecting tensor-shape errors, however, suffer from inherent limitations: static analysis methods often incur substantial overhead, while dynamic shape inference techniques tend to provide incomplete error detection.

In this paper, we present ARGUS, the first approach to effectively detect tensor-shape errors in deep learning models using concolic execution. Our core idea is to combine dynamic shape inference with static analysis through concolic execution of the target programs to comprehensively and effectively detect potential shape errors. We build a software prototype for ARGUS, and our evaluation results demonstrate that ARGUS can more effectively detect potential tensor-shape errors in deep learning models than the state-of-the-art approach while maintaining reasonable runtime efficiency.

Index Terms—Deep learning programs, Tensor-shape errors, Bug detection

I. INTRODUCTION

With the rapid development of deep learning technologies and frameworks, deep learning programs have been increasingly adopted in a wide range of safety-critical domains (*e.g.*, aircraft collision avoidance systems [1], disease diagnosis [2], and autonomous driving [3]). Given their considerable code sizes and complex logic, bugs are inevitable in these programs. Such bugs can lead to catastrophic consequences in safety-critical systems [4]. Therefore, detecting bugs in deep learning programs remains both urgent and essential.

Unfortunately, despite this urgency and essentiality, bugs in deep learning programs still manifest [5]. Among these bugs, tensor-shape errors emerge as one of the most frequent and critical categories of bugs in real-world deep learning programs [6]. Here, tensor-shape errors refer to a situation where the shapes of the relevant tensors do not satisfy the operator requirements. As an illustrative example, Fig. 1 presents a representative tensor-shape error occurring in model construction code, from the arguably most popular framework PyTorch [7]. Specifically, the `self.linear` (L4) multiplies the result of `self.lin2` (L5) with a weight tensor of a specific shape (8, 8) as an input. However, the output tensor x

```
class Model(torch.nn.Module):
2   def __init__(self):
3       super().__init__()
4       self.linear = torch.nn.Linear(8, 8)
5       self.lin2 = torch.nn.Linear(8, 1)
6
7   def forward(self, x):
8       x = self.lin2(x)
9       y_pred = self.linear(x)
10      return y_pred
11
```

Fig. 1: A tensor-shape error example from PyTorch.

of `self.lin2` has a shape of $(batch_size, 1)$, which is incompatible with the expected input shape of `self.linear`, leading to a tensor-shape error at runtime. Unfortunately, detecting such tensor-shape errors in an early development stage remains challenging because such errors may manifest only with specific inputs in dynamic computation graphs generated from PyTorch.

In response to this challenge, researchers have proposed static analysis to detect tensor-shape errors. The key idea of static analysis is to track and analyze tensor shape-related information in deep learning program source code *before* actually executing it [8]–[10]. When potential tensor-shape errors are identified, they are reported to end developers to facilitate early rectifications. Unfortunately, while static analysis is complete in theory, it incurs substantial computational overhead thus hindering its usage in practice, especially for large programs with considerable code sizes.

Another promising approach is dynamic shape error detection, which analyzes programs during execution to identify tensor-shape errors. Specifically, to enable program tracing, dynamic shape error detection normally rewrites tensor APIs to extract tensor shapes and to track shape changes [11]. Moreover, dynamic detection can leverage more concrete information than static analysis during execution, thus producing more elaborated error information [12]. Unfortunately, dynamic approaches, while valuable, lack comprehensiveness because they only cover and analyze the taken paths using the specific inputs. Consequently, dynamic approaches will miss those tensor-shape errors even in the taken paths if they are not triggered by the specific inputs.

* The corresponding author.

In this paper, we propose a concolic execution-based approach to effectively and comprehensively detect tensor-shape errors in deep learning models. Our core idea is to integrate dynamic shape inference into static analysis to enable concolic execution [13], which simultaneously maintains concrete and symbolic values during program execution. With symbolic values, we construct tensor shape constraints which are then solved by Satisfiability Modulo Theories (SMT) solvers, and we leverage solver outputs to comprehensively detect potential tensor-shape errors in deep learning models.

With this core idea, we take several steps to detect shape errors. First, we preprocess the target deep learning model code prior to detection by applying program slicing [14] techniques to minimize the code segments that need to be analyzed. We then conduct concolic execution on these preprocessed segments to generate and collect relevant runtime information and construct a set of model constraints. Finally, we employ a constraint-solving module to process the collected set of constraints. If a tensor-shape error is detected, the solver can also generate input tensor shapes that satisfy all collected constraints, assisting users in adjusting their input shapes accordingly. When the concolic execution does not reveal any concrete tensor-shape error, the solver computes input tensor shapes that could potentially lead to such errors, enabling comprehensive detection of possible tensor-shape error and preventing users from providing invalid inputs to the model.

We implemented a prototype for our approach, dubbed ARGUS, and conducted extensive experiments to evaluate its practical effectiveness and usability. The evaluation is performed with two datasets we constructed from the official PyTorch repository and tensor-shape error examples from Stack Overflow. Overall, ARGUS analyzed all test cases constructed from official PyTorch model examples in 13.75 ms on average, demonstrating ARGUS’s high execution efficiency. Meanwhile, ARGUS achieved a precision of 100% and a recall of 80% on the dataset collected from Stack Overflow, demonstrating its comprehensive capability in detecting tensor-shape errors in deep learning models.

In summary, this paper makes the following contributions:

- We propose the first approach to effectively detect tensor-shape errors in deep learning models leveraging a concolic execution.
- We design and implement a software prototype ARGUS to validate our approach.
- We conduct extensive evaluation to demonstrate ARGUS’s practical efficiency and effectiveness.

II. APPROACH

In this section, we present a detailed description of our approach to detect tensor shape errors. We begin with an overview of our approach (§ II-A), then detail the design of each component (§ II-B to § II-E), respectively.

A. Overview

We present an overview of ARGUS’s workflow in Fig. 2, comprising three key components: preprocessing, concolic

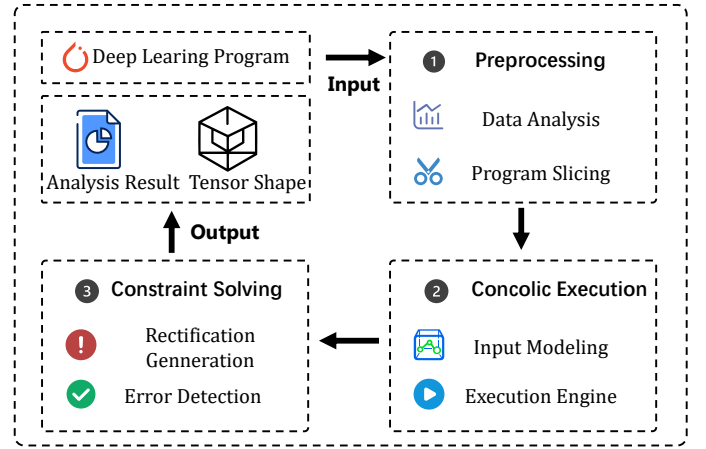


Fig. 2: An overview of ARGUS’s workflow.

execution and constraint collection, and constraint solving. First, the preprocessing module (1) takes as inputs the deep learning programs to minimize the code segments by applying program slicing techniques, and outputs stripped models. Second, the concolic execution (2) performs concolic execution on the preprocessed model by simultaneously maintaining both the concrete and symbolic states of input tensor shapes. This module constructs a set of constraints to accurately and comprehensively analyze potential tensor shape errors in the model, based on the semantics of each tensor operator. Finally, the constraint solving module (3) discharges the collected constraints and issues informative messages about tensor errors for subsequent analysis.

B. Preprocessing

The preprocessing module reduces overhead by minimizing the code analyzed. Production deep learning models often include extraneous functionalities like data logging and visualization. To avoid costly exploration of irrelevant paths, ARGUS leverages a static backward slicing technique starting from the model invocation point. This traces control and data dependencies, producing a significantly smaller, semantically equivalent code fragment containing only essential model construction and parameter preparation.

C. Concolic Execution

We employ concolic execution to detect tensor shape errors and collect tensor shape constraints on the tensor programs that are generated by program slicing from the preprocessing module. Specifically, the concolic execution comprises two components of input modeling and the execution engine, which are discussed next, respectively.

Input Modeling. Our core idea of concolic execution is to simultaneously maintain both concrete values and symbolic values of variables during tensor program execution, and update them in a synchronized manner as execution progresses. Specifically, ARGUS models all input parameters and intermediate variables of the target model by constructing a symbolic

Expression	$E ::= \mathcal{N} \mid \mathcal{B}$
Number	$\mathcal{N} ::= x \in \text{Var} \mid c \in \mathbb{Z}$ $\mid \mathcal{N} \oplus \mathcal{N}$
Numeric Op.	$\oplus ::= + \mid - \mid * \mid / \mid \dots$
Boolean	$\mathcal{B} ::= b \in \text{BVar} \mid \text{True} \mid \text{False}$ $\mid \mathcal{B} \wedge \mathcal{B} \mid \mathcal{B} \vee \mathcal{B} \mid \neg \mathcal{B}$ $\mid \mathcal{N} \ominus \mathcal{N}$
Relational Op.	$\ominus ::= = \mid < \mid \leq \mid \dots$

Fig. 3: The abstract syntax of symbolic expressions in ARGUS.

representation for each variable and maintaining a pair (u, v) where the meta-variable u and v represent the concrete value and symbolic value, respectively. As we focus on tensor shape evaluation in this work, ARGUS symbolically models each dimension of the tensor shape to obtain a symbolic shape representation for any tensor variables in the target program.

Execution Engine. ARGUS takes model construction code as its analysis target and represents it in a unified abstract syntax tree (AST) form, on which concolic execution is performed. We present a fragment syntax of ARGUS’s symbolic expressions E in Fig. 3. The syntax of E comprises both numerical expressions $\mathcal{N}$ and boolean expressions $\mathcal{B}$, where the former consists of variables x , numerical constants c , and binary operations $\oplus$, and the latter comprises boolean variables b , boolean constants (e.g., `True` and `False`), boolean operations (e.g., $\wedge$ and $\vee$), and relational operations $\ominus$.

Based on this abstract syntax, ARGUS combines concrete execution with symbolic execution by simultaneously maintaining concrete values and symbolic values. Specifically, we represent the abstract machine state model for concolic execution by a pair of map (σ, β) : $\sigma : x \mapsto u, \beta : x \mapsto v$, where the concrete map σ maps each variable x to a concrete value $u \in E$ and the symbolic map β each variable x is mapped to its corresponding symbolic value $v \in E$, respectively. These two forms of values are updated in a synchronized manner during program execution. In addition, based on the input tensor requirements of each operator, ARGUS incrementally collects tensor shape constraints during analysis.

We use $\mathcal{R}(E, \sigma, \beta)$ to formalize the concolic execution of E with machine state (σ, β) , as shown in Fig. 4. For operations like arithmetic $\oplus$ and relational $\ominus$, sub-expressions are evaluated to produce concrete and symbolic pairs concurrently. Boolean operations specifically yield results for both the program’s concrete domain (e.g., $u_1 \& u_2$) and the solver’s logic domain (e.g., $v_1 \wedge v_2$).

It is worth noting that, since our focus is on detecting tensor shape errors rather than errors in numerical computation, ARGUS does not actually execute the computational logic of the underlying operators during dynamic execution. Instead, ARGUS infers and propagates tensor shape information according to operator semantics and tensor broadcasting rules. This design decision significantly reduces the computational overhead of execution.

Concolic execution of loops and conditional branches poses

$$\begin{aligned}
\mathcal{R}(n, \sigma, \beta) &= (n, n) \\
\mathcal{R}(x, \sigma, \beta) &= (\sigma(x), \beta(x)) \\
\mathcal{R}(\mathcal{N}_1 \oplus \mathcal{N}_2, \sigma, \beta) &= \text{let } (u_1, v_1) = \mathcal{R}(\mathcal{N}_1, \sigma, \beta) \\
&\quad \text{let } (u_2, v_2) = \mathcal{R}(\mathcal{N}_2, \sigma, \beta) \\
&\quad \text{in } (u_1 \oplus u_2, v_1 \oplus v_2) \\
\mathcal{R}(\mathcal{N}_1 \ominus \mathcal{N}_2, \sigma, \beta) &= \text{let } (u_1, v_1) = \mathcal{R}(\mathcal{N}_1, \sigma, \beta) \\
&\quad \text{let } (u_2, v_2) = \mathcal{R}(\mathcal{N}_2, \sigma, \beta) \\
&\quad \text{in } (u_1 \ominus u_2, v_1 \ominus v_2) \\
\mathcal{R}(\mathcal{B}_1 \wedge \mathcal{B}_2, \sigma, \beta) &= \text{let } (u_1, v_1) = \mathcal{R}(\mathcal{B}_1, \sigma, \beta) \\
&\quad \text{let } (u_2, v_2) = \mathcal{R}(\mathcal{B}_2, \sigma, \beta) \\
&\quad \text{in } (u_1 \& u_2, v_1 \wedge v_2) \\
\mathcal{R}(\mathcal{B}_1 \vee \mathcal{B}_2, \sigma, \beta) &= \text{let } (u_1, v_1) = \mathcal{R}(\mathcal{B}_1, \sigma, \beta) \\
&\quad \text{let } (u_2, v_2) = \mathcal{R}(\mathcal{B}_2, \sigma, \beta) \\
&\quad \text{in } (u_1 \vee u_2, v_1 \vee v_2) \\
\mathcal{R}(\neg \mathcal{B}, \sigma, \beta) &= \text{let } (u, v) = \mathcal{R}(\mathcal{B}, \sigma, \beta) \\
&\quad \text{in } (!u, \neg v)
\end{aligned}$$

Fig. 4: Evaluation rules for symbolic expressions in ARGUS.

a challenge, because they easily lead to state explosion. To tackle this issue, we employ a strategy to leverage the concrete value σ of the condition expression $\mathcal{B}$ to determine the desired control flow, while recording the symbolic expression β of the condition as part of the path constraints. Our strategy has the benefit that, during constraint solving, the input tensor shapes returned by the solver satisfy the conditions imposed by the current execution path. Upon program execution completion, we obtain path condition like $\bigwedge_{i=1}^n \mathcal{P}_i$, where each $\mathcal{P}_i$, $1 \leq i \leq n$, represents the path condition constraint added at the i -th conditional branch. We argue that an important invariant of tensor shapes at a dedicated program point should be identical; otherwise, it constitutes a model design error and should trigger a warning. To guarantee this invariant, when handling branch constructs, we perform symbolic execution on all branches to obtain all possible tensor shapes. At a merge point, we introduce coercion constraints to enforce that the tensor shapes produced by different branches are consistent. Furthermore, when concolic execution encounters a conditional branch whose condition cannot be symbolically or concretely resolved, ARGUS proceeds by following the branch that satisfies the condition and inserts an explicit assertion at the corresponding source location to represent that the branch condition cannot be fully discharged at this point.

We collect constraints throughout the entire concolic execution process. Besides path condition constraints, we also gather the tensor shape constraints that must be satisfied to meet the semantics of corresponding deep learning operators. Specifically, a tensor shape constraint defines the conditions under which a deep learning program can execute correctly without triggering shape errors. For example, in a matrix multiplication operator $A_{m,k} \times B_{k,y}$, the inner dimensions

$$\begin{array}{c}
\frac{\Gamma(t_A) = \mathcal{S}_A \quad \Gamma(t_B) = \mathcal{S}_B \quad K = \max(\text{rank}(\mathcal{S}_A), \text{rank}(\mathcal{S}_B))}{\Gamma \vdash \text{element_op}(t_A, t_B) \rightsquigarrow \forall k \in [1, K]. (d_A(k) = d_B(k) \vee d_A(k) = 1 \vee d_B(k) = 1)} \quad (\text{ELEMENTOP}) \\
\\
\frac{\Gamma(t_A) = \mathcal{S}_A \quad \Gamma(t_B) = \mathcal{S}_B \quad \text{rank}(\mathcal{S}_A) = \text{rank}(\mathcal{S}_B) = 2}{\Gamma \vdash \text{mm}(t_A, t_B) \rightsquigarrow \mathcal{S}_A[1] = \mathcal{S}_B[0]} \quad (\text{MATMUL}) \\
\\
\frac{\Gamma(t_A) = \mathcal{S}_A}{\Gamma \vdash \text{reshape}(t_A, \mathcal{S}_{\text{target}}) \rightsquigarrow \prod \mathcal{S}_A = \prod \mathcal{S}_{\text{target}}} \quad (\text{RESHAPE})
\end{array}$$

Fig. 5: Constraint generation rules for representative operators.

k and x of the participating tensors must be consistent: $k = x$. Similarly, operators such as convolution, element-wise operations, and tensor reshaping impose explicit specific constraints on the number of dimensions and the size of each dimension.

To guide the tensor shape constraints generation, we formally define derivation rules for generating operator constraints. Given the huge volume of tensor operators in a deep learning library such as PyTorch, we next illustrate and highlight three representative rules in Fig. 5: element-wise operations, matrix multiplication, and tensor reshaping. As they represent typical kinds of operations in deep learning, derivation rules for other operators are similar. Specifically, we use the notation $\Gamma \vdash op \rightsquigarrow \mathcal{C}$ to denote that, under the context Γ (which maps variables to their shapes), the operator op generates the corresponding shape constraints $\mathcal{C}$.

For element-wise operations `element_op` (such as addition and subtraction), the corresponding operator constraints are generated by the element-wise rule (**ELEMENTOP**) in Fig. 5. Specifically, for each dimension, the two input tensors t_A and t_B must either have equal sizes (*i.e.*, $d_A(k) = d_B(k)$) or one of them must be equal to 1, in accordance with broadcasting rules. Matrix multiplication (**MATMUL**) requires the number of columns of the first matrix to match the rows of the second. Reshaping (**RESHAPE**) preserves the total item count. By continuously maintaining concrete and symbolic shapes during execution, sys dynamically accumulates these respective shape constraints to form a complete constraint set for the solver.

D. Constraint Solving

The constraints solving module processes the path conditions and tensor shape constraints generated during concolic execution, and discharges these constraints to judge whether they are satisfiable. We employ different constraint solving strategies depending on the concolic execution state, as Algorithm 1 presents.

Specifically, when we detect a tensor shape error during concolic execution (L_1), we generate a constraint Φ from the path constraint Φ_{path} and the tensor shape constraint Φ_{tensor} , and feed Φ to a constraint solving engine to discharge this constraint. The result shape from the solving engine specifies valid input tensor shapes that can satisfy both the path

Algorithm 1: Constraint-solving strategies.

Input: Path constraint Φ_{path} ; tensor-shape constraint Φ_{tensor} ; execution error flag err
Output: An input tensor shape assignment that satisfies the constraint

```

if  $\text{err} = \text{True}$  then
   $\Phi \leftarrow \Phi_{\text{path}} \wedge \Phi_{\text{tensor}};$ 
   $(r, \text{shape}) \leftarrow \text{Solve}(\Phi);$ 
  if  $r = \text{SAT}$  then
    return  $\text{shape};$ 
  else
    return UNSAT;
  end
else
   $\Phi \leftarrow \Phi_{\text{path}} \wedge \neg \Phi_{\text{tensor}};$ 
   $(r, \text{shape}) \leftarrow \text{Solve}(\Phi);$ 
  if  $r = \text{SAT}$  then
    return  $\text{shape};$ 
  else
    return NoCounterExample;
  end
end

```

constraints Φ_{path} and the tensor shape constraint Φ_{tensor} , thus serving as repair suggestions to end model developers to assist them in model debugging and bug rectification. In the absence of errors, we leverage constraint solvers to find input tensor shape that satisfies the path constraint Φ_{path} but violates the tensor shape constraint Φ_{tensor} , thereby uncovering potential shape errors in that dedicated path . Notably, the returned shape can be leveraged to generate a counterexample to trigger that potential tensor shape error.

E. Prototype Implementation

We have implemented a software prototype for ARGUS. We implemented ARGUS in Python to facilitate seamless integration with mainstream deep learning frameworks such as PyTorch and TensorFlow. The concolic execution module reuses part of the type analysis functionality from Elichika [15] and leverages `gast` as an intermediate representation to parse and traverse the model construction code. In our initial engineering efforts, we utilized Python’s built-in `ast`

TABLE I: Overhead evaluation and detection effectiveness comparison.

Model	ET(ms)	ST(ms)	ARGUS	Elichika [15]	Description
dcgan	3.91	20.1	✓	×	A GAN built with deep convolution operators for image generation.
mnist	0.6	8.3	✓	×	Classic digit-recognition neural network.
mnist_hogwild	1.17	6.35	✓	×	Digit-recognition neural network trained in parallel.
super_resolution	0.75	7.75	✓	×	Image super-resolution model using sub-pixel convolution layers.
alexnet	2.1	6.4	✓	×	One of the most classic image-classification neural networks.
time_sequence_prediction	12.17	20.35	✓	×	Time-series prediction model.
fast_neural_style	7.03	7.06	✓	×	A model for neural style transfer.
vae	1.63	2.35	✓	×	An improved implementation of Auto-Encoding Variational Bayes.
Total time	28.99	78.61	–	–	–

ET stands for execution time.
ST stands for solving time.

module, but found that `gast` provides better cross-version compatibility than the `ast` module through unifying syntax node differences across Python versions, thereby reducing the effort required for extensive branching logic in the analysis and lowering implementation complexity and maintenance costs.

We implemented the constraint-solving module by interfacing with Z3 [16], a powerful logic solver with many dedicated theories. Specifically, we utilized the linear arithmetic theory extensively because affine operations are ubiquitous in production deep learning libraries. We utilized Z3’s Python interface to facilitate the smooth integration into the deep learning ecosystem.

III. EVALUATION

To evaluate the effectiveness of ARGUS, we conduct extensive evaluations. Specifically, our evaluation aims to answer the following research questions:

RQ1: Usefulness. Can ARGUS effectively detect tensor shape errors in real-world deep learning programs?

RQ2: Overhead. ARGUS is designed to detect potential tensor shape errors in deep learning models before actual execution. An important question is whether the overall analysis overhead remains within an acceptable range.

All experiments were conducted on a Linux server with an Intel Core i7, 16GB RAM, and an RTX 3060.

A. Datasets

We constructed two datasets: a small-scale dataset of manually completed tensor shape error snippets from StackOverflow, and a large-scale dataset of representative deep learning models from the official PyTorch example repository.

B. RQ1: Usefulness

This section mainly investigates whether ARGUS can effectively detect tensor shape errors that commonly occur in real-world deep neural network development. Since the erroneous cases in the small-scale dataset are primarily collected from

real-world tensor shape errors, we conduct usefulness evaluation experiments on this dataset. We adopt precision, recall, and the F_1 score as the primary metrics for evaluation.

TABLE II: Evaluation results.

Total	$TP + FN$	TP	Precision	Recall	F_1
20	10	8	100%	80%	88.9%

The detailed evaluation results are shown in Table II. ARGUS evaluated all 20 deep learning programs in the small-scale dataset. Among the programs containing tensor shape errors, ARGUS successfully detected 8 of these errors. Of the remaining two undetected errors, one of the undetected errors was caused by using a tensor generated after the model had successfully executed to perform a subsequent matrix multiplication, resulting in a shape error. This type of error does not correspond to a shape error in the model itself, which is the primary focus of ARGUS, and therefore could not be detected. The other undetected error was due to complex control flow logic in the model, where branch conditions depend on concrete intermediate computation values. During detection, ARGUS does not actually execute these computations, which prevents the correct path constraints from being collected and leads to detection failure. Overall, ARGUS achieves high precision and recall on the small-scale dataset, demonstrating its effectiveness in detecting tensor shape errors in deep learning programs.

We also compared ARGUS with the state-of-the-art dynamic tensor type checking tool Elichika [15]. On the large-scale dataset, ARGUS successfully detected all potential tensor shape errors in the evaluated models, whereas Elichika failed to identify these problems. This demonstrates that our concolic execution-based approach is more effective than existing dynamic tensor type checking methods in detecting tensor shape errors in deep learning programs.

C. RQ2: Overhead

This section focuses on the additional performance overhead incurred by ARGUS when analyzing different neural networks in real-world projects. To evaluate this overhead, we conduct performance overhead experiments on the large-scale dataset. Table I summarizes the overhead statistics for each evaluated model.

As shown in the table, ARGUS completes the detection task for all evaluated models efficiently, achieving an average detection time of 13.75 ms. The incurred overhead is modest and does not impose a significant time burden on users, indicating that ARGUS can be well integrated into practical model development and debugging workflows.

IV. RELATED WORK

Tensor-shape error is one of the most common sources of bugs in deep learning programs, and a number of studies have explored techniques to detect such errors at different stages of program execution.

Static analysis attempts to detect tensor-shape errors prior to execution [9], [10], [17]–[19]. These approaches track shapes using type systems or constraint solving but typically suffer from path explosion and high analysis overhead, which limits their scalability.

Conversely, dynamic approaches validate tensor shapes under given inputs during execution [11], [12], [20]. While faster and precise for the covered path, they only assess correctness for the specific input shapes used, potentially leaving errors triggered by other input shapes undetected.

V. CONCLUSION

We have presented ARGUS, a concolic-execution based approach to detect potential tensor-shape errors in deep learning models. Detecting such errors comprehensively is challenging: purely static techniques can incur substantial overhead, while dynamic shape inference can only determine whether a shape is incorrect based on the specific input shapes provided. Using a concolic execution approach, we collect and solve the constraints that must be satisfied by the model's tensor shapes, thereby exposing potential shape errors in the model. We confirmed the effectiveness of ARGUS on nontrivial PyTorch programs, showing millisecond-level overhead on representative example models and strong detection effectiveness on real-world mismatch cases (e.g., 100% precision and 80% recall on our small dataset). We believe our approach can effectively help deep learning researchers detect potential hidden tensor-shape errors in their models, thereby assisting them in developing models more reliably.

REFERENCES

- [1] K. D. Julian, M. J. Kochenderfer, and M. P. Owen, "Deep neural network compression for aircraft collision avoidance systems," *Journal of Guidance, Control, and Dynamics*, vol. 42, no. 3, pp. 598–608, Mar. 2019.
- [2] S. Liu, S. Liu, W. Cai, S. Pujol, R. Kikinis, and D. Feng, "Early diagnosis of alzheimer's disease with deep learning," in *2014 IEEE 11th International Symposium on Biomedical Imaging (ISBI)*. Beijing, China: IEEE, Apr. 2014, pp. 1015–1018.
- [3] C. Chen, A. Seff, A. Kornhauser, and J. Xiao, "Deepdriving: Learning affordance for direct perception in autonomous driving," in *2015 IEEE International Conference on Computer Vision (ICCV)*. Santiago, Chile: IEEE, Dec. 2015, pp. 2722–2730.
- [4] N. Kalra and S. M. Paddock, "Driving to safety: How many miles of driving would it take to demonstrate autonomous vehicle reliability?" *Transportation Research Part A: Policy and Practice*, vol. 94, pp. 182–193, Dec. 2016.
- [5] 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*. Amsterdam Netherlands: ACM, Jul. 2018, pp. 129–140.
- [6] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan, "A comprehensive study on deep learning bug characteristics," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Tallinn Estonia: ACM, Aug. 2019, pp. 510–520.
- [7] "How to solve size mismatch error in python (pytorch) - stack overflow," <https://stackoverflow.com/questions/61186998/how-to-solve-size-mismatch-error-in-python-pytorch>.
- [8] J. Dolby, A. Shinnar, A. Allain, and J. Reinen, "Ariadne: Analysis for machine learning programs," in *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. Philadelphia PA USA: ACM, Jun. 2018, pp. 1–10.
- [9] S. Lagouvardos, J. Dolby, N. Grech, A. Antoniadis, and Y. Smaragdakis, "Static analysis of shape in tensorflow programs," *LIPICs, Volume 166, ECOOP 2020*, vol. 166, pp. 15:1–15:29, 2020.
- [10] H. Y. Jho, S. Kim, W. Song, K. Park, D. Lee, and K. Yi, "A static analyzer for detecting tensor shape errors in deep neural network training code," in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. Pittsburgh Pennsylvania: ACM, May 2022, pp. 337–338.
- [11] S. Verma and Z. Su, "Shapeflow: Dynamic shape interpreter for tensorflow," Nov. 2020.
- [12] M. Hattori, S. Sawada, S. Hamaji, M. Sakai, and S. Shimizu, "Semi-static type, shape, and symbolic shape inference for dynamic computation graphs," in *Proceedings of the 4th ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. London UK: ACM, Jun. 2020, pp. 11–19.
- [13] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Computing Surveys*, vol. 51, no. 3, pp. 1–39, May 2019.
- [14] M. Weiser, "Program slicing," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 4, pp. 352–357, Jul. 1984.
- [15] M. Hattori, S. Sawada, S. Hamaji, M. Sakai, and S. Shimizu, "Semi-static type, shape, and symbolic shape inference for dynamic computation graphs," in *Proceedings of the 4th ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. London UK: ACM, Jun. 2020, pp. 11–19.
- [16] L. De Moura and N. Björner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, C. R. Ramakrishnan, and J. Rehof, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, vol. 4963, pp. 337–340.
- [17] M. Hattori, N. Kobayashi, and R. Sato, "Gradual tensor shape checking," in *Programming Languages and Systems*, T. Wies, Ed. Cham: Springer Nature Switzerland, 2023, vol. 13990, pp. 197–224.
- [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)*. Melbourne, Australia: IEEE, Nov. 2021, pp. 55–66.
- [19] D. Wu, B. Shen, Y. Chen, H. Jiang, and L. Qiao, "Automatically repairing tensor shape faults in deep learning programs," *Information and Software Technology*, vol. 151, p. 107027, Nov. 2022.
- [20] Y. Wang, J. A. H. López, U. Nilsson, and D. Varró, "Using run-time information to enhance static analysis of machine learning code in notebooks," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*. Porto de Galinhas Brazil: ACM, Jul. 2024, pp. 497–501.