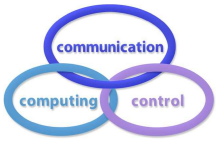
INTERNATIONAL JOURNAL OF COMPUTERS COMMUNICATIONS & CONTROL
Online ISSN 1841-9844, ISSN-L 1841-9836, Volume: 20, Issue: 5, Month: October, Year: 2025
Article Number: 6885, https://doi.org/10.15837/ijccc.2025.5.6885



CCC Publications

Optimizing Symbolic Execution Path Exploration with a Transfer Learning-Based Strategy

T. Sun, D. Zhu, L. He, D. Zhang

Te Sun
School of Cyberspace Science and Technology
Beijing Jiaotong University, Beijing, China
suntepeking@163.com

Dongqing Zhu
Beijing Jiaotong University, Beijing, China
22120480@bjtu.edu.cn

Lianying He
Beijing Jiaotong University, Beijing, China
22120375@bjtu.edu.cn

Dalin Zhang*
Beijing Jiaotong University, Beijing, China
*Corresponding author: dalin@bjtu.edu.cn

Abstract

Symbolic execution is an important software analysis technique, but it faces challenges such as path explosion, which leads to a reduction in efficiency. Existing path exploration strategies, such as Random State Search, typically exhibit poor adaptability to real-world programs and lack effective path selection strategies. To address these challenges, this paper proposes a Transfer Learning-based Symbolic Execution Path Exploration Strategy, TLS (Transfer Learning Search). We adopt a transfer learning method based on functional classification to optimize existing symbolic execution strategies. Specifically, real-world programs are classified according to their functional characteristics, and transfer learning is applied by freezing partial layers of existing neural networks with training sets from each program family that better reflect its features. Multiple models are trained based on different training sets to adapt to various program families. Experimental results show that this strategy solves the problem of insufficient training data for real-world programs. Compared to traditional heuristic methods such as random-path (rps) and random-state (rss) strategies, this approach significantly improves instruction coverage and branch coverage on specific program families. For example, in the Grep program test, branch coverage increased by approximately fifteen percentage points, generating more test cases. This approach provides a new and effective solution to the adaptability problem of symbolic execution for complex programs.

Keywords: Transfer Learning, Symbolic Execution, Path Exploration Strategy, Symbolic State.

1 Introduction

As an analysis technique for enhancing software quality, symbolic execution replaces program variables with symbolic inputs, performing a simulated execution of symbolic sets on variables, statements, and expressions during runtime. This concept was first introduced by King et al. in 1975 [16]. From the early heuristic symbolic execution strategies, such as the random-path strategy (rps), to the current machine learning-based methods, like Learch [12], symbolic execution has gradually evolved into a crucial tool in program analysis. Today, it is widely applied in various security-related tasks, such as verifying hardware designs [33] and detecting cache timing leaks [11].Its primary advantage lies in its ability to explore different program paths, generating concrete inputs for each path while identifying potential errors in the program. As a result, symbolic execution is extensively used in software analysis and testing [1, 30]. The main goal of symbolic execution tools is to generate a test suite that maximizes code coverage of the program's statements [10]. By systematically analyzing every branch and path of a program, symbolic execution can uncover potential vulnerabilities, errors, or abnormal behavior.

Since the introduction of symbolic execution, the path explosion problem [32] has remained one of the primary challenges in this field. For instance, during symbolic execution, each conditional branch generates two or more possible execution paths. Consequently, as the number of branches increases, the program's execution paths grow exponentially. Symbolic execution struggles to perform a comprehensive analysis when faced with an overwhelming number of paths, a difficulty that becomes especially apparent when dealing with complex real-world programs [27].To address this challenge, engineers urgently need an efficient selection and execution mechanism to identify the most promising states, guiding symbolic execution tools to prioritize the most relevant paths. This would enable higher code coverage in a shorter time while avoiding unnecessary redundant states. However, relying solely on the real-time attributes of states for selection is insufficient, as it is often impossible to predict whether a given state will significantly improve code coverage within a reasonable cost. As a result, exploring more strategic selection criteria is crucial to solving this issue.Path exploration strategies in symbolic execution have traditionally been limited to manually designed heuristic methods. In an effort to optimize state selection, various researchers have proposed heuristic approaches based on different metrics [19, 24], aiming to enhance the state selection process in symbolic execution. While heuristic methods can alleviate the path explosion problem to some extent, their lack of effective predictive capabilities means that state selection remains overly constrained. Consequently, these methods tend to favor specific metric-based regions of a program, making it difficult to explore other areas.Given this analysis, if a state selection mechanism with dynamic predictive capabilities could be established—one that combines multiple metric attributes—symbolic execution could more effectively improve code coverage in test cases and help uncover additional security vulnerabilities.

To address the issues above, the existing path exploration strategy, Active Learning Search (ALS) [35], transforms the state selection problem into an active learning problem, proposing a symbolic execution path exploration strategy based on active learning. Experiments show that ALS can achieve higher code coverage and detect more security violations than existing heuristic algorithms, effectively alleviating the path explosion problem in symbolic execution. However, ALS has shown suboptimal results on certain real-world programs, mainly due to poor model adaptability. This is caused by functional and structural differences between programs and the limited size of real-world training sets, which leads to ALS's predictive model struggling to accurately assess the reward values of symbolic states in new programs, thereby impacting the efficiency and effectiveness of path exploration. The features learned by the model during training may not directly apply to programs with different characteristics, resulting in decreased code coverage and test case generation capability in these programs.

To address this issue, improving model adaptability is crucial. Therefore, this study adopts a transfer learning-based symbolic execution path exploration strategy, Transfer Learning Search (TLS). This strategy classifies programs with similar functionalities and utilizes the training set obtained from the symbolic execution of an existing model and one program within a category to perform transfer learning. In other words, the existing model is trained through transfer learning with the newly acquired training set according to a specific strategy, resulting in a new model. The resulting model is then applied to the symbolic execution of the remaining programs in the same category.

During the transfer learning process, the existing neural network model undergoes partial layer freezing, while the newly obtained training set is used to retrain the network. This allows the model to retain most of its existing knowledge while better capturing the features of the selected class of programs. Moreover, this approach helps address the class imbalance problem, where the training set obtained from the symbolic execution of the GNU coreutils suite is much larger than that of real-world programs. As a result, it enhances the performance of the original ALS model within program families related to the training programs.

In this study, we implemented the active transfer learning-based symbolic execution path exploration strategy on the KLEE symbolic execution engine. Experimental results demonstrate that the TLS method can enhance ALS model performance in terms of instruction coverage, branch coverage, and the number of generated test cases for certain program families within KLEE symbolic execution. However, these improvements are

not consistent or significant across all cases, making it challenging to draw universally applicable conclusions. The main contributions of this study are as follows:

1. We propose the Transfer Learning Search (TLS), a symbolic execution path exploration strategy that combines active learning and transfer learning. Through functionality-based transfer learning, TLS enables the model to learn from and adapt to functionally related programs, improving model efficiency and adaptability, and successfully integrates TLS within the KLEE engine.
2. This study frames the adaptability challenge in symbolic execution strategies as a neural network transfer learning problem based on functionality classification. By leveraging the commonalities between different programs, TLS enhances the model's generalization capability and adaptability.
3. This study addresses the issue of insufficient training data in real-world programs. Class imbalance biases the model towards coreutils data patterns, neglecting the characteristics of real programs, which in turn reduces both performance and generalization ability. To mitigate this, we employ a transfer learning strategy that adjusts parameters of models trained on coreutils data to better adapt to the unique features of real programs, alleviating the data imbalance problem.
4. The proposed method was validated on four real-world programs from different functional categories, showing significant improvements in specific program families. For example, in function execution of the Grep program, TLS achieved an 11.0% increase in instruction coverage, a 12.1% increase in branch coverage, and successfully generated two test cases (whereas ALS generated none). However, the experiment also revealed that the effect of this method on some program families did not meet expectations, highlighting certain limitations. For instance, in the Objcopy program test, TLS underperformed, with its two coverage indicators falling behind those of most other strategies.

2 Related Work

2.1 Transfer learning

Transfer learning is a machine learning technique that centers on transferring knowledge, features, or model parameters learned from a source domain to a target domain to enhance the model's performance in the target domain [22, 28, 36]. Transfer Learning (TL) aims to improve performance on a current task by relating it to previously completed related tasks. By identifying common knowledge between the source and target tasks, transfer learning facilitates knowledge transfer, providing a faster solution [23].

The working principle of transfer learning is as follows [13]: A domain D consists of a feature space X and its marginal probability distribution $P(X)$, where $X = \{x_1, \ldots, x_n\}$ represents the set of features. Given a domain $D = \{X, P(X)\}$, a task (T) is defined as $T = \{Y, f(\cdot)\}$, where Y is the label space and $f(\cdot)$ is the objective predictive function. A task is learned from the pairs (x_i, y_i), where $x_i \in X$ and $y_i \in Y$. In transfer learning, given a source domain D_S with a source task T_S and a target domain D_T with a target task T_T, the goal of transfer learning is to improve the learning of the target predictive function $f_T(\cdot)$ in D_T by utilizing the knowledge from D_S and T_S .

By retaining the rich information from the source domain possessed by the pre-transfer model, transfer learning can reduce the need for large amounts of labeled data, enhance the model's generalization ability, and accelerate the training process. Transfer learning has been widely applied in fields such as computer vision and natural language processing [15, 34].

T. Kaur et al. [14] employed various pre-trained neural network models (such as AlexNet, ResNet50, and GoogLeNet DCNN) to evaluate their effectiveness in pathological brain image classification. To meet specific image classification requirements, the study replaced the final layers of these models and trained and tested them using benchmark datasets such as Harvard, clinical, and Figshare. Test results indicate that AlexNet, combined with transfer learning, achieved the best performance in a shorter time, surpassing traditional machine learning methods and conventional CNN models.

The widely-used GPT models also employ a transfer learning strategy [25].Alec Radford et al. demonstrated that by performing generative pre-training on a large-scale, diverse corpus of unlabeled text and then fine-tuning discriminatively on specific tasks, significant performance improvements can be achieved across various natural language processing tasks. Compared to traditional methods, this approach enables effective transfer learning through task-aware input transformations without extensive modifications to the model architecture, thereby significantly simplifying the model adaptation process.

The transfer learning-based symbolic execution path exploration model used in this study is constructed using a functionality-based transfer learning approach [15], as shown in Figure 1. This method transfers the pre-trained model to the target domain, employing strategies such as training only the feature extraction layers (the initial layers) of the network to fine-tune parameters, thereby enhancing the model's predictive capability on programs within similar categories to the target domain.

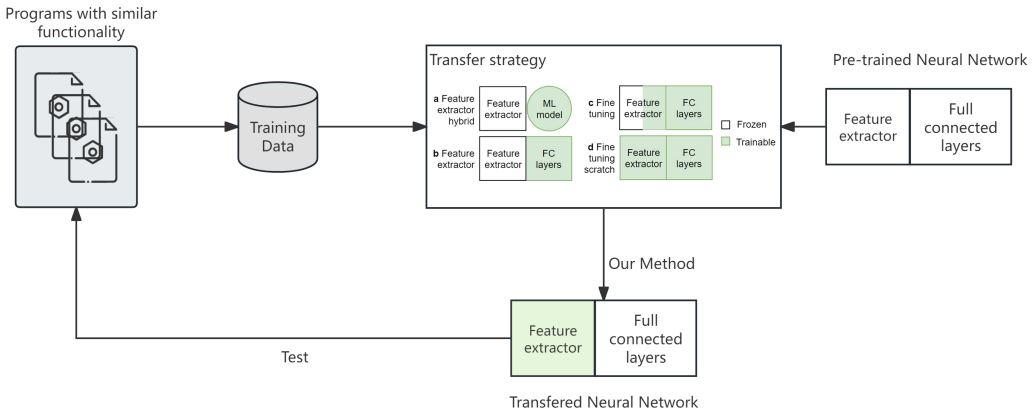


Figure 1: Transfer learning process of TLS

2.2 Symbolic Execution

As a widely used software analysis technique, symbolic execution offers powerful analytical capabilities. The core principle involves tracing the program's execution path and generating symbolic constraints by representing the program's variables and input values as symbolic values (instead of specific constants or variables). By simulating the program's execution and observing its behavior under different input conditions, it becomes possible to deeply analyze each execution path and uncover potential errors and vulnerabilities. However, traditional symbolic execution methods often face challenges related to path explosion and constraint solving [1]. To address the path explosion problem, numerous studies have proposed various solutions. One prominent approach focuses on optimizing path selection algorithms, improving the path search strategy in symbolic execution to enhance overall efficiency.

For instance, KLEE [5] provides multiple path search strategies to address the path explosion problem. These strategies include depth-first search (DFS), breadth-first search (BFS), random state search, and random path search, all aimed at optimizing the path exploration efficiency of symbolic execution.

Recently, machine learning techniques have demonstrated significant improvements in enhancing the testing efficiency of symbolic execution, indirectly validating the feasibility of using machine learning to optimize symbolic execution. Specifically, Learch [12] applies supervised learning to optimize path selection in symbolic execution, selecting the most optimal states for path exploration. Sooyoung Cha et al. [6] proposed a method to automatically generate heuristic rules for symbolic execution search through offline learning.

In this study, we propose a state selection strategy based on active transfer learning to address the challenge of insufficient training data in real-world programs. Traditional symbolic execution methods often struggle with path explosion and insufficient test coverage when applied to complex software systems. While existing improvements—such as heuristic search and machine learning-driven path selection—have shown promise, they typically rely on large amounts of training data. However, the diversity of real-world programs and the scarcity of labeled training data make these approaches difficult to generalize across different software systems.To overcome this limitation, our approach integrates active transfer learning into symbolic execution, enabling the strategy to migrate across different but related programs and efficiently adapt to new target programs. This significantly enhances test coverage and test case generation. Our method leverages function-based transfer learning to optimize existing symbolic execution strategies, ultimately forming a Transfer Learning-based Symbolic Execution (TLS) strategy.Experimental results demonstrate that, compared to traditional symbolic execution methods, our approach achieves higher instruction and branch coverage across multiple program testing tasks and generates a greater number of effective test cases. As a result, it usually delivers robust testing performance across diverse software applications.The following sections will provide a detailed discussion of the implementation, theoretical analysis, and experimental validation of our approach.

3 Transfer Learning-based Symbolic Execution Path Exploration Strategy (TLS)

3.1 Description of the TLS Approach

In symbolic execution path exploration, the existing Active Learning Search (ALS) strategy has already achieved two main objectives:

1. Combining various heuristic strategies and prediction model advantages, ALS prioritizes path exploration

of states that can achieve higher coverage within unit time, thereby generating high-quality test cases.

2. Based on the principle of active learning, ALS constructs a feedback mechanism that guides the prediction model to focus more on symbolic states with high reward values, using prediction results to inform model updates.

TLS was proposed mainly to address issues observed in ALS:

1. The active learning strategy (ALS) model, trained on a dataset from the GNU coreutils suite, exhibits poor adaptability when applied to certain real-world programs. This lack of adaptability is mainly due to significant differences between GNU coreutils and real-world programs, particularly in terms of code complexity and program structure. GNU coreutils programs generally feature simple functional structures with stable code logic, which can be fully covered by symbolic execution [12, 18]. In contrast, real-world programs often include numerous third-party libraries, dependencies, multithreading operations, and other complex features, making their control flow more difficult to predict and resulting in a substantial increase in the number of branches.To address this issue, the idea of incorporating real-world training data into the existing dataset was explored to improve model adaptability. However, experimental results indicated that this approach did not yield the expected improvements. The main reason is that simply adding a small amount of real-world data to a training set primarily based on GNU coreutils is insufficient to bridge the gap in code characteristics and distribution between the two types of programs, as demonstrated below.

2. As illustrated in Figure 2, GNU coreutils, being a standardized set of command-line tools, has a small codebase and simple structure, making it easier for symbolic execution to cover code paths and generate a large number of samples. In contrast, real-world programs are more complex, requiring more resources for symbolic execution, which results in a smaller training set [17]. Therefore, as shown, the training set obtained from symbolic execution on the GNU coreutils suite is significantly larger than that from real-world programs.This discrepancy leads to a class imbalance problem, where certain classes in the training set have substantially more samples than others. Class imbalance causes the model to favor learning overrepresented classes, while underrepresented classes are overlooked, leading to model bias, reduced generalization ability, and an increased risk of false positives and negatives. As a result, the ALS model struggles to accurately handle minority classes in practical applications, such as path exploration in certain real-world programs. In summary, simply merging datasets does not resolve the adaptability issue of the original ALS strategy.

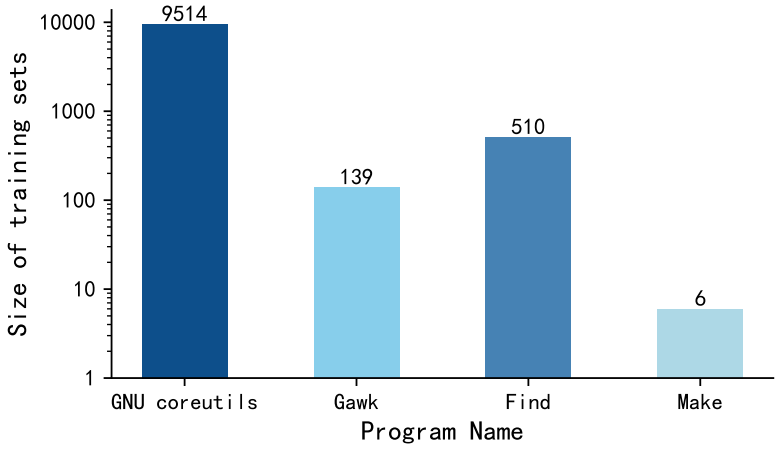


Figure 2: Differences in Training Set Sizes Across Programs

To address the issues mentioned above, we introduced a functionality-based transfer learning approach. Our method is grounded in homogeneous transfer learning and utilizes a feature transformation strategy [29]. Transfer learning can be classified into two categories: homogeneous transfer learning and heterogeneous transfer learning, depending on the differences between the source and target domains. In this experiment, since the target domain (real-world programs) and the source domain (GNU coreutils programs) share the same feature space, our method falls under homogeneous transfer learning. We employed a feature transformation approach as our transfer strategy [36], where feature transformation involves identifying shared latent features (such as latent topics) between the source and target domains, which serve as a bridge for knowledge transfer.

In our experiments, different categories of programs exhibit distinct feature representations (i.e., the weights of various parameters vary). Our study employs a function-based categorization approach for transfer learning, which assumes that real-world programs with similar functionality share similar latent features. By training

a neural network on the dataset of a single program, our model can learn new feature representations that align with these latent characteristics, enabling it to adapt to other programs with similar functionality. This strategy allows the neural network to learn the latent feature representation of one program and generalize it to other functionally similar programs.

Since programs within the same functional category typically share similar internal structures and control flows, we combined the model trained using the ALS strategy with programs in specific functional domains. Through transfer learning, the model is fine-tuned to better adapt to the unique characteristics of these types of programs. The fine-tuned model is then applied to other functionally similar programs within the same domain, improving performance in symbolic execution. This approach leverages functional similarity, enabling the model to capture shared features across programs, thus covering more code paths and enhancing the efficiency and accuracy of symbolic execution.

Compared to directly using a model trained on GNU coreutils, the functionality-based transfer learning strategy effectively addresses the class imbalance issue and boosts the model's generalization ability within the same functional domain. As a result, it offers greater stability and accuracy in symbolic execution, improving adaptability to complex real-world scenarios.

In the TLS strategy, the neural network model obtained through the ALS algorithm is applied to functionally classified program families. For each program family with similar functionality, one program is selected as the source for the training set, on which heuristic-based symbolic execution is performed to generate the training data. A transfer strategy is then applied: layer freezing is used to lock certain layers of the neural network model, preserving knowledge from prior tasks, while fine-tuning is performed only on specific layers. This fine-tuning is done using the new training set, resulting in the transferred neural network.

Overall, this study introduces a novel path exploration strategy by integrating established transfer learning algorithms with the existing active learning-based symbolic execution path exploration strategy. The core of the strategy is a fine-tuned machine learning model that learns path characteristics specific to a program category, enabling better adaptation to such programs. This active transfer learning-based path exploration strategy not only retains the advantage of active learning, which reduces the need for extensive labeling of large datasets, but also addresses class imbalance and enhances model adaptability to real-world environments through transfer learning. The strategy demonstrates both theoretical innovation and practical applicability, effectively tackling the complexity and diversity challenges in real-world program analysis. The workflow for updating the model in the transfer learning component of TLS is shown in Figure 3.

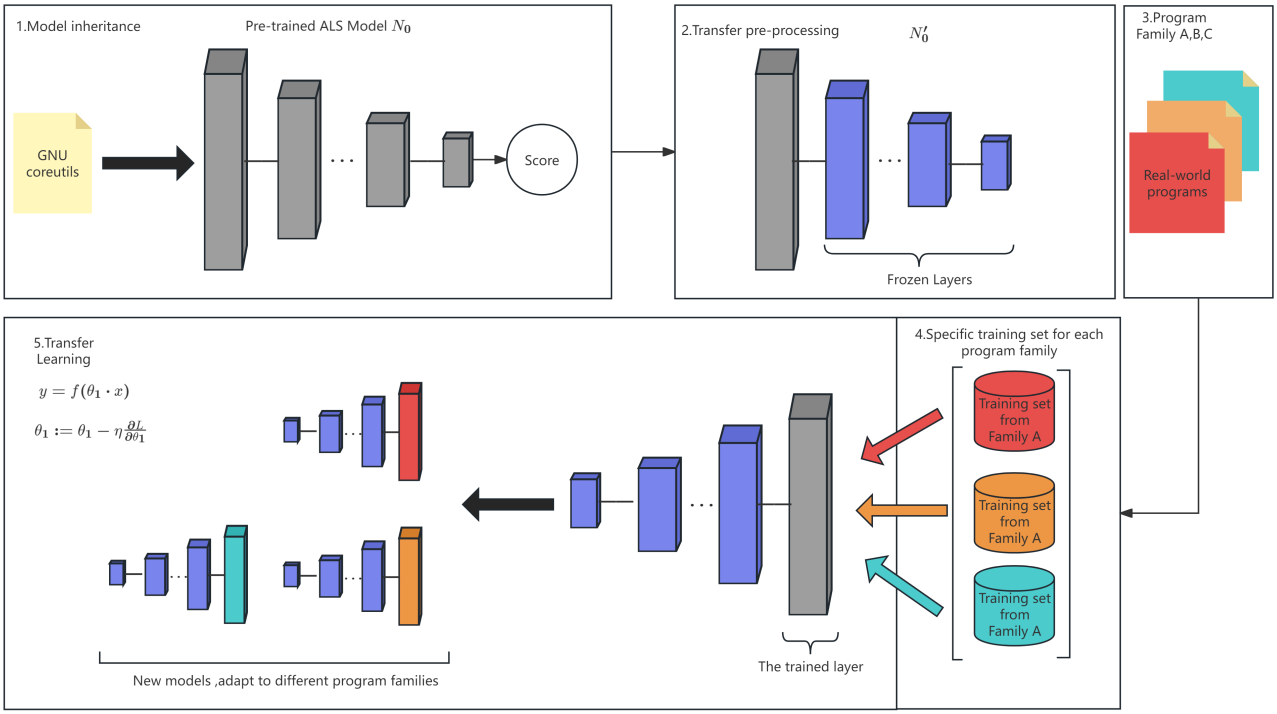


Figure 3: Workflow of TLS

The specific implementation steps for each stage are as follows:

1. **Model Inheritance**: From the ALS strategy, we obtain a neural network N_0 that has already acquired knowledge for effective symbolic execution path exploration on the GNU coreutils. For this existing neural

network, we have a set of parameters $\{\theta_1, \ldots, \theta_n\}$, where n is the total number of layers in the neural network.

2. **Transfer Processing**: For the parameter set $\{\theta_1, \ldots, \theta_n\}$, a partial fine-tuning strategy is applied, where only θ_1 remains trainable while freezing the remaining layers $\{\theta_2, \ldots, \theta_n\}$. This produces a new neural network N_0', ensuring that the knowledge learned by the network is largely retained.
3. **Functional Classification of Programs**: Programs are categorized based on functional similarity into several classes, such as A, B, C, etc.
4. **Obtaining Training Sets**: For each class of programs, such as A, one program (e.g., a_1) is selected to undergo heuristic symbolic execution, generating a new training dataset T_{a_1}.
5. **Training the Transferred Model**: The transferred model N_0' is trained on the training set T_{a_1} to produce a new neural network model N_A, which is better adapted to program class A. This model is then tested on the remaining programs in class A (e.g., a_2 through a_m).

   The specific transfer process is as follows: During the transfer process, the feedforward calculation formula is:

$$y = f(\theta_1 \cdot x),$$

   where:

   - $x \in \mathbb{R}^n$ represents the input vector,
   - $\theta_1 \in \mathbb{R}^{m \times n}$ represents the weight matrix of the first layer,
   - $f(\cdot)$ denotes the activation function of the first layer, and
   - $y \in \mathbb{R}^m$ represents the output of this layer.

   During training, only the parameter θ_1 is optimized, while the parameters of the other layers remain unchanged. Therefore, the loss function $L(y, \hat{y})$ is differentiated only with respect to θ_1, and the update rule is:

$$\theta_1 = \theta_1 - \eta \frac{\partial L}{\partial \theta_1},$$

   where:

   - η is the learning rate, and
   - $\frac{\partial L}{\partial \theta_1}$ is the gradient of the loss function with respect to θ_1.

3.2 Feature Selection

In the reward prediction component of TLS, various feature values are extracted from the symbolic states of existing test cases. In this study, these features are determined based on the heuristic path exploration strategies already incorporated in the symbolic execution tool KLEE, along with key properties relevant to symbolic execution. By selecting these feature values, we retain some of the advantages of heuristic algorithms [5] while achieving improved symbolic execution performance. The feature values selected in this study are categorized into two groups.

1. **Static Features**
   The static features of symbolic states selected in this study include metrics such as the current call stack size and the bag-of-words representation of path constraints. These features are generated before symbolic execution begins and can thus be directly obtained from the original state, as follows:

   - **stack**: The size of the current call stack.
   - **constraints**: The bag-of-words representation of path constraints.

2. **Dynamic Features**
   During symbolic execution, symbolic states exhibit changing characteristics, which are dynamically collected as the program executes. We refer to these as dynamic features, specifically:

   - **successors**: The number of successors of the current basic block.
   - **genTestCases**: The number of test cases generated so far.
   - **coverage**: The number of instructions and lines covered.
   - **depth**: The number of branches executed.
   - **cp__inst__count**: The number of instructions executed within a function.

- **inst_count**: The total execution count of instructions.
- **instSinceCovNew**: The number of instructions executed since the last new instruction was covered.
- **subpath**: The number of times the state subpath has been visited.

3.3 TLS Generation

The goal of reward prediction is to score the symbolic states generated by the program, determining whether to select a particular state for further path exploration during symbolic execution. This process helps efficiently filter out states worth analyzing, optimizing both test coverage and quality.

TLS uses the feedforward neural network (FNN) from ALS to build the prediction model. A feedforward neural network (FNN) propagates input data through multiple layers, applying weights at each layer and performing nonlinear transformations on these weighted values using activation functions, ultimately producing an output. This process maps the input data from one vector space to another. Essentially, an FNN abstracts, compresses, or amplifies features of the input data through stacked layers of linear transformations and nonlinear activations, capturing complex relationships and underlying patterns. This makes FNNs highly effective for tasks such as prediction and classification, especially in fields like image recognition and natural language processing. As a result, FNNs have become one of the most fundamental and widely used models in deep learning [8, 9, 31].

Based on these advantages, this study selects FNN as the prediction model. Algorithm 1 describes the specific framework of the TLS model generation process, including the two main stages of generating the training set and training the model.

Algorithm 1 TLS Model Training Algorithm: Example with Training Set from Class A Programs

Input: Initial neural network N_0, initial program set P

Output: Path exploration model TLS

1: $N_0 \leftarrow N_{\text{ALS}}$ ▷ Obtain the initial prediction model from ALS
2: $N_0' \leftarrow N_0$ ▷ Prepare for transfer learning
3: programSet(A, B, ...) $\leftarrow P$ ▷ Classify the initial program set by functionality
4: $a_1 \in A$ ▷ Select one program a_1 from class A
5: $T_{a_1} \leftarrow a_1$ ▷ Obtain the training set
6: $N_a \leftarrow \text{trainingfeed}(T_{a_1})$ ▷ Perform transfer learning to obtain the new model
7: TLS $\leftarrow N_a$
8: **return** TLS
9: Test TLS on the remaining programs in class A: $A \setminus \{a_1\}$

1. **Generating the Training Set**
   To efficiently generate test cases for program files, we first use the symbolic execution tool KLEE to perform symbolic execution on each program file individually. During this process, KLEE's built-in heuristic path exploration strategy is employed to extract symbolic states and their key features. By analyzing each symbolic state, we generate corresponding feature vector representations F, providing a basis for subsequent test case generation and path selection.

   Next, tools such as gcov are used to calculate the reward value for each symbolic state, which is then used as a label. To comprehensively evaluate the overall impact of each symbolic state within test cases, this study considers both code coverage and execution efficiency when calculating reward values. Specifically, the reward value is determined by the ratio of the total code coverage achieved by each symbolic state (including its derived branch states) to the total time consumed during execution. This design ensures that the reward value effectively reflects the combined performance of symbolic states in terms of both test coverage and resource consumption.

   As shown below:

$$\text{Reward}(\text{state}) = \frac{\sum_{t \in \text{tests}(\text{state})} \text{Coverage}(t)}{\sum_{m \in \text{states}(\text{state})} \text{StateTime}(m)}$$

   In this formula, *state* represents a symbolic state, *Coverage* and *StateTime* denote the code coverage and total time consumed by this state in a test case, respectively. *tests* represents the generated test paths, and *states* represents all generated symbolic states. *Reward* represents the code coverage efficiency of the state in unit time.

Table 1: The basic information about the coreutils suite and the real-world programs

Program Name	Version	Functionality	Symbolic Input Configuration
coreutils	8.31	A set of commonly used system utilities	-sym-args 0 1 10 -sym-args 0 2 2 -sym-files 1 8 -sym-stdin 8
diff	3.7	Compares file contents and shows differences	–sym-args 0 2 2 A B –sym-files 2 50
grep	3.6	Searches for text patterns in files	–sym-args 0 2 2 –sym-arg 10 A –sym-files 1 50
find	4.7.0	Searches for files and directories	–sym-args 0 3 10 –sym-files 1 40 –sym-stdin 40 –sym-stdout
readelf	2.3.6	Displays structural information of ELF files	-a A –sym-files 1 100
make	4.3	Automation tool for compilation	-n -f A –sym-files 1 40
gawk	5.1.0	Text processing language for data scanning and processing	-f A B –sym-files 2 50 –sym-stdout
objcopy	2.3.6	Converts object file formats or modifies binaries	–sym-args 0 2 2 A –sym-files 1 100 –sym-stdout

To optimize the state selection strategy, we prioritize selecting symbolic states with the highest reward values, guiding symbolic execution path exploration to maximize test coverage and execution efficiency.

2. **Model Training**
The model training process begins with preparation for transfer on the neural network N_0 obtained from the ALS strategy. For the parameter set θ, only θ_1 is set as a trainable parameter, while all remaining parameters are frozen, resulting in the neural network N_0', which will be used for training.

Next, programs within the test program set are classified based on functionality into several categories, such as A, B, C, etc. Subsequently, one program from each category is selected, and a heuristic algorithm is applied to obtain its training set. This training set is then used to train N_0', producing a neural network that is applicable to all programs with similar functionality.

At this point, the functionality-based transfer learning process is complete.

4 Experiment and analysis

4.1 Dataset

The experimental platform for this study is a Docker container running on an Intel(R) Xeon(R) Gold 6130 CPU @ 2.10GHz with 128GB of RAM, using the Ubuntu 18.04 operating system. The coreutils suite (version 8.31) from the GNU software collection provides a variety of tools for file, shell, and text operations [4]. These tools are diverse and structurally complex, covering various types of software and systems, and are widely used to evaluate the effectiveness of testing tools based on symbolic execution. As a result, they have become the standard test suite for symbolic execution tools like KLEE, offering a reliable benchmark and test environment for assessing and comparing the performance of symbolic execution techniques in different application scenarios.

In this study, part of the program files from the coreutils suite, along with three real-world program files, were used to train the model. Four additional real-world program files were used to test the model.

The same input configuration commands were applied to all packages in the coreutils suite. Table 1 provides detailed information on the program projects used in this study's experiments, including the binary file size and functionality of each program package.

4.2 Experimental Design

In this study, the selected programs were systematically categorized based on their functional characteristics to more clearly present the role of each tool in different application scenarios. First, `gawk` and `grep` were classified as text processing tools, primarily used for pattern matching, scanning, and processing text, with powerful text handling capabilities. Second, `find` and `diff` were categorized as file search and comparison tools: `find` is used for searching files or directories in the file system, while `diff` compares differences in file contents. Both tools play an essential role in file management and version control. Finally, `make`, `readelf`, and `objcopy` were grouped as compilation and binary file manipulation tools, responsible for automated program building and compilation, parsing, and modifying binary file structures. These tools are widely used in program development and debugging. This classification method not only highlights the core functionality of each program but also provides a theoretical basis and practical guidance for subsequent experiments and performance evaluations.

In this study, six baseline methods were selected, most of which are inherent heuristic strategies within the symbolic execution tool KLEE [3, 4]. These methods include the random-path (rps) strategy, random-state (rss) strategy, nurs:cpicnt (nurc) strategy, nurs:depth (nurd) strategy, sgs:1 strategy, learch strategy, and the foundational ALS strategy developed in prior research. Each of these strategies selects states based on different attributes of symbolic states, and each has its unique advantages.

The random-path search strategy (rps) performs path exploration by constructing a binary tree, where the leaf nodes represent symbolic states to be processed, and the internal nodes correspond to previously explored states. The random-state search strategy (rss), on the other hand, is primarily influenced by the number of symbolic states, randomly selecting a state from the list of pending states. Consequently, rss only requires a simple random selection operation, allowing it to maintain relatively linear execution efficiency and storage demand, even with a large number of symbolic states.

The non-uniform search strategies, nurs:cpicnt and nurs:depth, evaluate and sort all pending states based on specific criteria: instruction counts for nurs:cpicnt and path depths for nurs:depth. These strategies offer higher time complexity but relatively low space complexity, scaling linearly with the number of pending states. This design increases computational overhead while ensuring more precise path selection.

The time complexity of TLS can be broadly divided into the complexity of feature extraction and that of the machine learning model, yielding a total time complexity of $O(nh^2 + fn)$. With frozen layers, $h = 1$. Compared to the baseline methods, disregarding the time complexity inherited from ALS, TLS has a lower time complexity due to the frozen layer, making it more efficient than baseline methods.

The space complexity of TLS includes the storage requirements for symbolic states and the model itself. The space complexity for storing symbolic states is $O(fn)$, and the model storage complexity is $O(\text{ALS})$, where ALS refers to the number of inherited parameters. Thus, the total space complexity of TLS is $O(fn) + O(\text{ALS})$.

In summary, compared to the baseline methods and the existing ALS method, TLS introduces higher complexity in both time and space for optimizing symbolic execution path exploration. However, through effective sample selection and model updating, TLS significantly enhances the efficiency of symbolic execution path exploration and improves adaptability to various complex real-world program files.

The TLS strategy in this study combines the strengths of the aforementioned strategies to some extent. The experimental validation of TLS primarily focuses on three aspects:

1. **Validation of Code Coverage Capability of TLS.** This part of the study evaluates the symbolic execution path exploration model based on active transfer learning to determine whether, compared to other search strategies, TLS can more effectively select states with higher reward values in most cases, thus achieving improved code coverage.

2. **Validation of TLS's Ability to Generate Test Cases.** In this part of the research, we tested the symbolic execution model based on active transfer learning, focusing on whether TLS can more effectively select states that generate test cases meeting path constraints, compared to other search strategies, thereby increasing the number of generated test cases.

3. **Comparison Between TLS and the Original ALS Strategy.** This aspect aims to highlight the innovation and strengths of this study. Through experimental validation of TLS, we explore its advantages in enhancing learning performance and transfer efficiency. Additionally, this study addresses any potential limitations and challenges of TLS, providing valuable insights and guidance for future research.

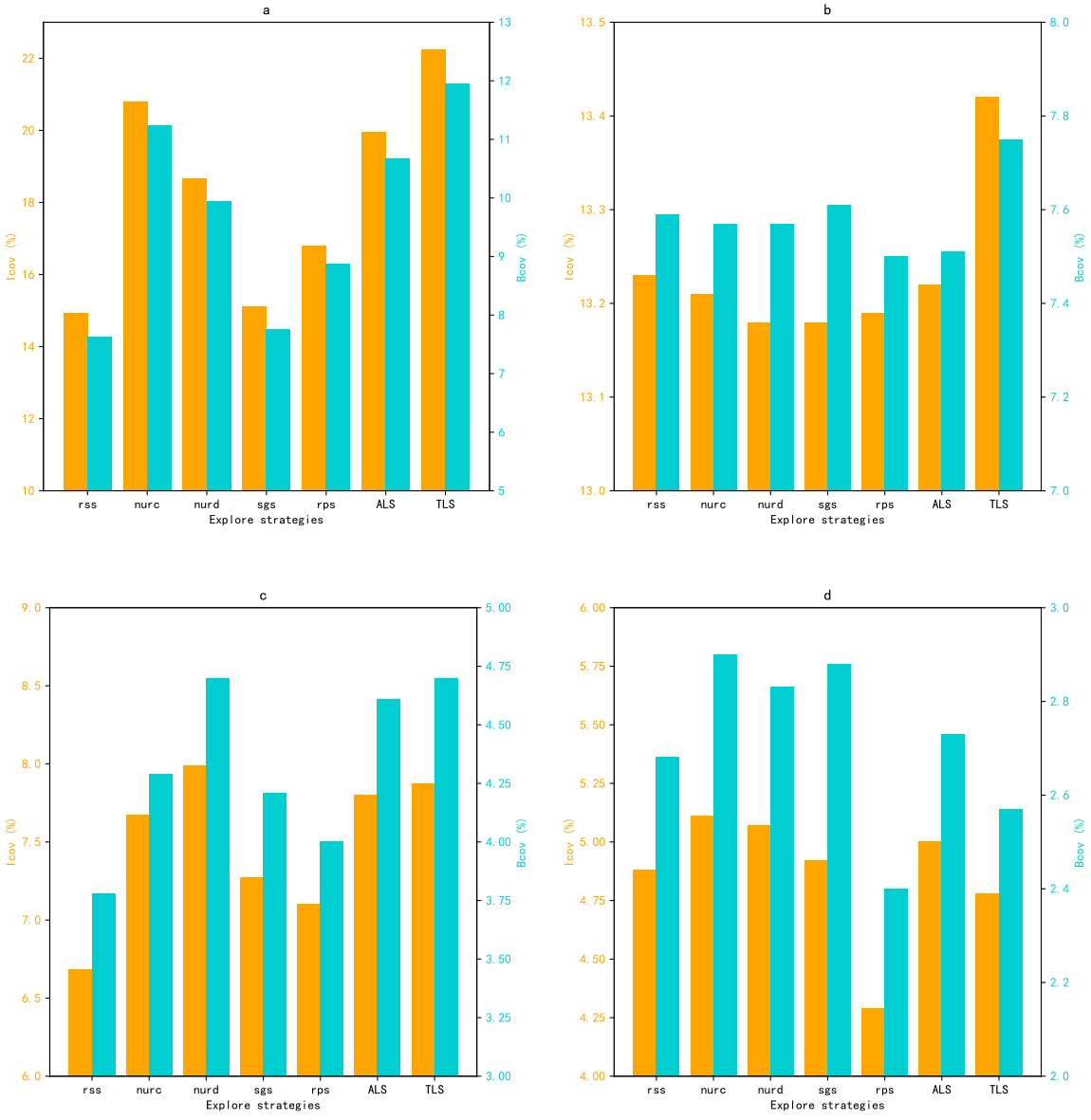
4.3 Validation of TLS Code Coverage Capability

To validate the code coverage capability of TLS, a series of experiments were conducted to assess the code coverage achieved by test cases generated through the symbolic execution path exploration strategy. The experiments used a functionality-based transfer learning framework, with three categories of real-world programs as test subjects: one category containing 1 program, another with 1 program, and the third with 2 programs. For each program, different search strategies were applied under two experimental conditions: a short-duration test (10 minutes) and a long-duration test (1 hour). Each experiment was repeated five times, and the instruction coverage (ICov) and branch coverage (BCov) metrics were recorded. The average values of these metrics were calculated for comparative analysis.The 10-minute test aimed to evaluate the efficiency of each strategy over a short period, while the 1-hour test assessed the strategy's potential for more extensive coverage. After completing the experiments, an in-depth analysis was performed on the three program categories to determine the extent to which TLS improved code coverage. Figure 4 shows the code coverage achieved within 10 minutes under different search strategies, and the detailed experimental data is provided in Table 2. These results offer a clear comparison of the short-term performance of each strategy.

Table 2: ICov and BCov Rates for 4 Real-World Programs Symbolically Executed for 10 Minutes with Different Search Strategies

Program	Coverage	rss	nurc	nurd	sgs	rps	ALS	TLS
Grep	ICov (%)	14.92	20.81	18.66	15.13	16.80	19.97	22.24
	BCov (%)	7.63	11.24	9.94	7.76	8.88	10.68	11.95
Diff	ICov (%)	13.23	13.21	13.18	13.18	13.19	13.22	13.42
	BCov (%)	7.59	7.57	7.57	7.61	7.50	7.51	7.75
Readelf	ICov (%)	6.68	7.67	7.99	7.27	7.10	7.80	7.87
	BCov (%)	3.78	4.29	4.70	4.21	4.00	4.61	4.70
Objcopy	ICov (%)	4.88	5.11	5.07	4.92	4.29	5.00	4.77
	BCov (%)	2.68	2.90	2.83	2.88	2.40	2.73	2.57

Figure 4: (a) Ten-minute test results of the model on the grep program after transfer training on the gawk training set.(b)Ten-minute test results of the model on the diff program after transfer training on the find training set.(c)(d) Ten-minute test results of the model on the readelf and objcopy programs after transfer training on the make training set.



In the 10-minute symbolic execution experiments conducted on four real-world programs, the TLS strategy achieved the highest instruction coverage and branch coverage in two of the programs. For example, in the Grep program test, TLS achieved 22.24% instruction coverage and 11.95% branch coverage, compared with the suboptimal strategy, it increased by 6.87% and 6.32% respectively; in the Diff program test, it achieved

Table 3: ICov and BCov Rates for 4 Real-World Programs Symbolically Executed for 1 hour with Different Search Strategies

Program	Coverage	rss	nurc	nurd	sgs	rps	ALS	TLS
Grep	ICov (%)	14.92	20.81	18.72	15.13	16.81	19.97	22.63
	BCov (%)	7.63	11.26	9.96	7.76	8.88	10.68	12.30
Diff	ICov (%)	13.24	13.24	13.31	13.22	13.01	13.24	13.51
	BCov (%)	7.64	7.62	7.69	7.63	7.45	7.56	7.83
Readelf	ICov (%)	6.68	7.94	8.19	7.27	7.43	8.03	7.99
	BCov (%)	3.78	4.59	5.00	4.21	4.47	4.92	4.90
Objcopy	ICov (%)	4.88	5.11	5.07	4.92	4.29	5.00	4.77
	BCov (%)	2.68	2.90	2.83	2.88	2.40	2.73	2.58

13.42% instruction coverage and 7.75% branch coverage, compared with the suboptimal strategy, it increased by 1.44% and 1.84% respectively. These values were higher than those of other strategies, indicating that TLS demonstrated superior performance on these two programs and was able to maintain excellent symbolic execution efficiency in a short period. In the Readelf program test, TLS obtained the second-best results, with instruction coverage only 1.5% lower than the best solution, and branch coverage equal to the best strategy, showing performance very close to the optimal solution. However, in the Objcopy program, TLS's performance was below expectations, with both coverage metrics lower than those of most other strategies.

Overall, the functionality-based transfer learning strategy significantly improved TLS's test coverage in some programs, demonstrating TLS's advantage in real-world applications: achieving high coverage within a limited timeframe and optimizing the effectiveness of symbolic execution. Nevertheless, the results on certain individual programs revealed some uncertainty in performance.

In Table 3, we present the results of a one-hour symbolic execution test conducted on four real-world programs to compare the coverage performance of different search strategies. With extended execution time, the performance of each strategy becomes clearer, allowing us to assess whether they have reached their performance limits.

The experimental results show that for Grep and Diff, which previously achieved relatively optimal results, the TLS strategy continued to improve both instruction coverage and branch coverage, compared with ALS, the results were improved by 11.37% and 11.90% respectively. In contrast, the traditional heuristic methods, which were suboptimal, showed little to no further progress. This indicates that TLS demonstrates greater potential for coverage improvement compared to traditional strategies. Additionally, in the symbolic execution of Readelf, although TLS did not surpass the best solution, its coverage remained close to the optimal strategy(about 2%), with only a small percentage difference. This shows TLS's competitiveness and adaptability.

Notably, in the Objcopy symbolic execution experiment, the coverage rates for TLS remained almost identical between the 1-hour and 10-minute runs, suggesting that its potential for this program may have reached a plateau, with no additional improvement observed. This implies that, for Objcopy, extending the duration of TLS's application did not yield further success. Overall, TLS demonstrated its superiority in symbolic execution across most test programs, but it also highlighted some performance limitations in specific scenarios.

In summary, the TLS strategy showed significant potential and advantages in symbolic execution testing, particularly in achieving optimal instruction and branch coverage in Grep and Diff. These results not only showcase TLS's efficiency in symbolic execution over short durations but also emphasize its adaptability across different program contexts, thanks to functionality-based transfer learning. Although its performance was somewhat limited in Readelf and Objcopy, the overall effectiveness of TLS affirms its viability as a robust symbolic execution strategy.

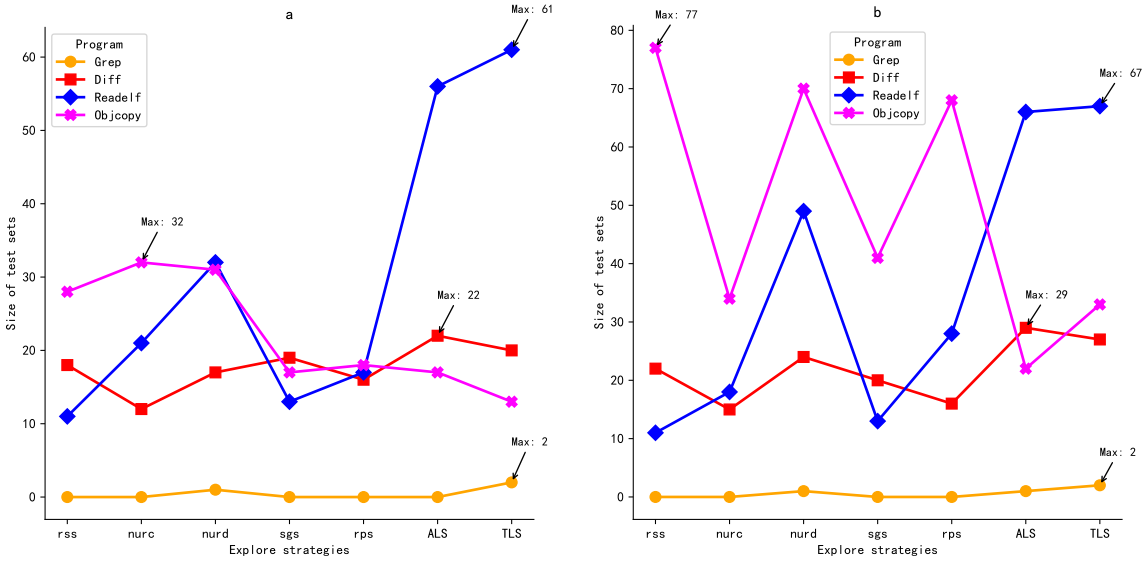
4.4 Validation of TLS Test Case Generation Capability

In this part of the study, a series of experiments were conducted on four real-world programs, with five rounds of testing performed for both 10-minute and 1-hour durations. The average values were calculated to evaluate the effectiveness of different path search strategies in generating test cases. Figure 5 compares the test cases generated by the different strategies, with detailed data recorded in Table 4. The results demonstrate that TLS delivers exceptional performance in test case generation for the Grep, Diff, and Readelf programs. Notably, TLS excels particularly in the Grep and Readelf programs, significantly outperforming heuristic algorithms.

Table 4: Number of Test Cases Generated for 4 Real-World Programs Symbolically Executed for 10 Minutes and 1 Hour with Different Search Strategies

Execution Time	Program	rss	nurc	nurd	sgs	rps	ALS	TLS
10 min	Grep	0	0	1	0	0	0	2
	Diff	18	12	17	19	16	22	20
	Readelf	11	21	32	13	17	56	61
	Objcopy	28	32	31	17	18	17	13
1 hour	Grep	0	0	1	0	0	1	2
	Diff	22	15	24	20	16	29	27
	Readelf	11	18	49	13	28	66	67
	Objcopy	77	34	70	41	68	22	33

Figure 5: (a)Number of test cases for 4 real-world programs running for 10 minutes under different search strategies.(b)Number of test cases for 4 real-world programs running for 1 hour under different search strategies.



In the Grep program, TLS generated two test cases within both the 10-minute and 1-hour execution periods, outperforming other strategies, particularly the heuristic algorithms (e.g., the RSS strategy and others, which failed to generate any test cases). This result demonstrates that TLS can efficiently generate meaningful test cases for Grep in a relatively short time. For the Diff program, TLS produced 20 and 27 test cases in the 10-minute and 1-hour intervals, respectively, which, although slightly fewer than ALS (22 and 29 test cases), still outperformed other strategies, highlighting TLS's competitiveness in test case generation for Diff.

The performance of TLS in test case generation for the Readelf program was particularly impressive, with 61 and 67 test cases generated within the 10-minute and 1-hour periods, respectively—significantly surpassing other heuristic strategies. In comparison, ALS generated 66 test cases in 1 hour, slightly fewer than TLS, underscoring TLS's significant advantage for this program.

However, TLS's performance on the Objcopy program was relatively poor when compared to the heuristic algorithms. In the 10-minute test, TLS generated only 13 test cases, far fewer than nurc (32). While the number of test cases generated by TLS increased to 33 after 1 hour, it still lagged behind RSS (77), indicating that TLS is less effective for Objcopy, particularly when compared to the heuristic algorithms.

Overall, TLS demonstrated strong performance in the Grep and Readelf programs, particularly excelling in test case generation for Readelf. However, its performance on Objcopy was weaker compared to several heuristic algorithms. This suggests that while TLS shows promise for certain programs, further optimization is needed for cases like Objcopy.

4.5 Effectiveness Analysis of TLS in the Active Learning Framework

To further highlight the innovation and effectiveness of this study, this section compares the experimental results of TLS with those of the previous ALS method developed in our lab. Both studies involved experiments for validating code coverage and test case generation effectiveness. When tested on the same set of real-world

programs, it was observed that the TLS model, optimized through targeted transfer learning, significantly enhanced the adaptability of the ALS strategy. In most cases, TLS demonstrated superior performance within specific real-world environments, showcasing its potential for broader applicability and efficiency in symbolic execution.

For instance, in the Grep program, TLS significantly increased instruction coverage (ICov) by +13.3% and branch coverage (BCov) by +15.2% in short-duration tests, generating more test cases (2 cases) compared to ALS. In long-duration tests, TLS also enhanced ICov and BCov and increased the number of generated test cases. This indicates that the model learned by the TLS strategy better adapts to the specific characteristics of the Grep program. On the Diff program, TLS also improved ALS's performance in terms of coverage and test case generation. For the Readelf program, although coverage remained unchanged, TLS still outperformed ALS in terms of test case generation, further demonstrating its transfer learning capability.

However, transfer learning does not always succeed. In the Objcopy program, TLS failed to improve the performance of ALS and even resulted in a slight decline. This suggests that while TLS performs well for most programs, there are limitations to its transferability in certain environments, highlighting the need for further optimization. By refining the transfer learning process, such as improving the selection of source migration data, we aim to address this issue.

In summary, the TLS model significantly enhanced the performance of the ALS strategy in most cases through transfer learning, particularly for the Grep, Diff, and Readelf programs. However, further research and optimization are necessary for specific programs, such as Objcopy.

From the comparative analysis of these experimental results, several key factors can explain these findings. First, the ALS model was trained on the GNU Coreutils suite, which may lack the adaptability required when applied to real-world programs. By employing functionality-based transfer learning, the ALS model can be better tailored to specific functional programs, thereby boosting its performance. However, the success of transfer learning heavily depends on the accuracy of functional classification, and there is currently no standardized method for assessing the functional similarity between programs. This uncertainty can lead to situations where the transferred model underperforms. Overall, while functionality-based transfer learning has the potential to significantly improve the ALS model's performance on specific programs, achieving greater stability and reliability will require the development of a more precise functional classification approach to minimize the risk of ineffective transfers.

5 Conclusion

This study addresses the challenges faced by current symbolic execution techniques and the limitations of mainstream path exploration strategies by proposing a transfer learning-based symbolic execution path exploration strategy, TLS. The uniqueness of this strategy lies in two main aspects:

1. TLS introduces a functionality-based transfer learning mechanism that enables the generic ALS model to adapt more effectively to specific real-world programs, broadening the scope of transfer learning applications. By classifying programs based on their functionality, TLS better captures the unique characteristics and behavioral patterns of target programs, enhancing the model's adaptability. This mechanism not only boosts performance in specific application scenarios but also increases the flexibility and adaptability of the general model in complex and diverse environments.

2. TLS combines the strengths of multiple heuristic search strategies with machine learning models, allowing for a comprehensive evaluation of symbolic states from various perspectives. Leveraging the KLEE platform, TLS automates the labeling of symbolic state reward values through a reward calculation method, eliminating the need for manual expert labeling. This approach significantly enhances path exploration efficiency and reduces the overall testing effort.

Experiments conducted on real-world programs demonstrate that, given the same symbolic execution time, TLS generally outperforms heuristic algorithms in both code coverage and test case generation capabilities. Furthermore, TLS shows varying degrees of improvement over ALS, with targeted transfer learning enabling the TLS model to achieve superior performance on multiple real-world programs (such as Grep, Diff, and Readelf), particularly in generating test cases and increasing coverage, thus significantly enhancing the original capabilities of ALS. For example, when testing the Grep program, TLS improved the two coverage rates by 6.87% and 6.32% respectively compared with the suboptimal strategy; when testing the Diff program, the two coverage rates were improved by 1.44% and 1.84% respectively compared with the suboptimal strategy.

However, the effectiveness of transfer learning heavily depends on the accuracy of functionality classification. In this study, functionality classification is based on the similarity between source and target programs. Yet, there is currently no scientific and systematic standard for evaluating program functionality similarity. Existing classification methods often rely on superficial program characteristics (such as application domains), without

fully accounting for the underlying logic of the programs. This imprecise classification can cause performance fluctuations in the transferred model, and, in some cases, even result in a decline in performance, as demonstrated in the Objcopy program, where TLS underperformed compared to ALS. This issue highlights the negative impact of inadequate functional similarity assessment, emphasizing the need for careful consideration of the relationship between source and target tasks in transfer learning. To address this, a more scientific classification approach is required—one that incorporates dynamic program characteristics, data flow analysis, and control flow analysis, ensuring greater accuracy and effectiveness in model transfer.

In conclusion, while functionality-based transfer learning introduces a promising approach to optimizing the ALS model and enhances its adaptability across various real-world programs, further work is necessary to ensure the stability of TLS's improvements in efficiency. Establishing a more rigorous and scientific functionality classification method is essential to minimizing potential negative effects during the transfer process and ensuring more effective transfer learning. Future research should focus on developing systematic classification standards and integrating multi-dimensional program characteristic analyses to minimize the risk of poor transfer outcomes, ultimately achieving more efficient transfer learning.

The long-term objective of this study is to develop a general symbolic execution model capable of adapting to different types of programs across diverse domains. Future efforts will aim at optimizing existing security vulnerability detection tools and improving their performance in complex programs by refining the symbolic execution path exploration strategy. These advancements will pave the way for more efficient, stable, and widely applicable symbolic execution technology, driving the development of automated software testing and security detection tools.

Funding

The APC was funded by R&D center "Cercetare Dezvoltare Agora" of Agora University.

Author contributions

The authors contributed equally to this work.

Conflict of interest

The authors declare no conflict of interest.

References

[1] Baldoni, R.; Coppa, E.; D'Elia, D.C.; Demetrescu, C.; Finocchi, I. (2018). *A survey of symbolic execution techniques*, ACM Computing Surveys (CSUR), 51(3), 1–39, 2018.

[2] Budd, S.; Robinson, E.C.; Kainz, B. (2021). *A survey on active learning and human-in-the-loop deep learning for medical image analysis*, Medical Image Analysis, 71, 102062, 2021.

[3] Burnim, J.; Sen, K. (2008). *Heuristics for scalable dynamic test generation*, 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, IEEE, 2008.

[4] Busse, F.; Nowack, M.; Cadar, C. (2020). *Running symbolic execution forever*, Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2020.

[5] Cadar, C.; Dunbar, D.; Engler, D.R. (2008). *Klee: unassisted and automatic generation of high-coverage tests for complex systems programs*, OSDI, 8, 209–224, 2008.

[6] Cha, S.; Hong, S.; Lee, J.; Oh, H. (2018). *Automatically generating search heuristics for concolic testing*, Proceedings of the 40th International Conference on Software Engineering, 2018.

[7] Cohn, D.; Atlas, L.; Ladner, R. (1994). *Improving generalization with active learning*, Machine Learning, 15, 201–221, 1994.

[8] Eldan, R.; Shamir, O. (2016). *The power of depth for feedforward neural networks*, Conference on Learning Theory, PMLR, 2016.

[9] Fine, T.L. (2006). *Feedforward neural network methodology*, Springer Science & Business Media, 2006.

[10] Godefroid, P.; Klarlund, N.; Sen, K. (2005). *DART: Directed automated random testing*, Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, 2005.

[11] Guo, S.; Wu, M.; Wang, C. (2018). *Adversarial symbolic execution for detecting concurrency-related cache timing leaks*, Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2018.

[12] He, J.; Sivanrupan, G.; Tsankov, P.; Vechev, M. (2021). *Learning to explore paths for symbolic execution*, Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, 2021.

[13] Hosna, A.; Merry, E.; Gyalmo, J.; Alom, Z.; Aung, Z.; Azim, M.A. (2022). *Transfer learning: a friendly introduction*, Journal of Big Data, 9(1), 102, 2022.

[14] Kaur, T.; Gandhi, T.K. (2020). *Deep convolutional neural networks with transfer learning for automated brain image classification*, Machine Vision and Applications, 31(3), 20, 2020.

[15] Kim, H.E.; Cosa-Linan, A.; Santhanam, N.; Jannesari, M.; Maros, M.E.; Ganslandt, T. (2022). *Transfer learning for medical image classification: a literature review*, BMC Medical Imaging, 22(1), 69, 2022.

[16] King, J.C. (1976). Symbolic execution and program testing, *Communications of the ACM*, 19(7), 385–394, 1976.

[17] Kurian, E.; Briola, D.; Braione, P.; et al. (2023). *Automatically generating test cases for safety-critical software via symbolic execution*, Journal of Systems and Software, 199, 2023, 111629.

[18] Kuznetsov, V.; Kinder, J.; Bucur, S.; Candea, G. (2012). *Efficient state merging in symbolic execution*, ACM Sigplan Notices, 47(6), 193–204, 2012.

[19] Li, Y.; Su, Z.; Wang, L.; Li, X. (2013). *Steering symbolic execution to less traveled paths*, ACM SigPlan Notices, 48(10), 19–32, 2013.

[20] Liu, W.; Zhang, H.; Ding, Z.; Liu, Q.; Zhu, C. (2021). *A comprehensive active learning method for multiclass imbalanced data streams with concept drift*, Knowledge-Based Systems, 215, 106778, 2021.

[21] Mohamad, S.; Sayed-Mouchaweh, M.; Bouchachia, A. (2018). *Active learning for classifying data streams with unknown number of classes*, Neural Networks, 98, 1–15, 2018.

[22] Neyshabur, B.; Sedghi, H.; Zhang, C. (2020). *What is being transferred in transfer learning?*, Advances in Neural Information Processing Systems, 33, 512–523, 2020.

[23] Pan, S.J.; Yang, Q. (2009). *A survey on transfer learning*, IEEE Transactions on Knowledge and Data Engineering, 22(10), 1345–1359, 2009.

[24] Păsăreanu, C.S.; Rungta, N. (2010). *Symbolic PathFinder: symbolic execution of Java bytecode*, Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering, 2010.

[25] Radford, A. (2018). *Improving language understanding by generative pre-training*, 2018.

[26] Settles, B. (2009). *Active learning literature survey*, University of Wisconsin-Madison Department of Computer Sciences, 2009.

[27] Siddiqui, J.H.; Khurshid, S. (2012). *Scaling symbolic execution using ranged analysis*, ACM Sigplan Notices, 47(10), 523–536, 2012.

[28] Sobolu, R.; Stanca, L.; Bodog, S. A. (2023). *Automated Recognition Systems: Theoretical and Practical Implementation of Active Learning for Extracting Knowledge in Image-based Transfer Learning of Living Organisms*, International Journal of Computers Communications & Control, 18(6), 2023.

[29] Weiss, K.; Khoshgoftaar, T. M.; Wang, D. D. (2016). *A survey of transfer learning*, Journal of Big Data, 3, 1–40, 2016.

[30] Wei, G.; Jia, S.; Gao, R.; et al. (2023). *Compiling parallel symbolic execution with continuations*, 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), IEEE, 2023, pp. 1316-1328.

[31] Wong, K.; Dornberger, R.; Hanne, T. (2024). *An analysis of weight initialization methods in connection with different activation functions for feedforward neural networks*, Evolutionary Intelligence, 17(3), 2024, pp. 2081-2089.

[32] Xie, T.; Tillmann, N.; De Halleux, J.; Schulte, W. (2009). *Fitness-guided path exploration in dynamic symbolic execution*, 2009 IEEE/IFIP International Conference on Dependable Systems & Networks, IEEE, 2009.

[33] Zhang, R.; Deutschbein, C.; Huang, P.; Sturton, C. (2018). *End-to-end automated exploit generation for validating the security of processor designs*, 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), IEEE, 2018.

[34] Zhu, Z.; Lin, K.; Jain, A.K.; Zhou, J. (2023). *Transfer learning in deep reinforcement learning: A survey*, IEEE Transactions on Pattern Analysis and Machine Intelligence, 2023.

[35] Zhu, D.; Zhang, J.; He, L.; Wang, R.; Liu, J.; Zhang, D. (2024). *Path Exploration Strategy Based on Active Learning for Symbolic Execution*, Submitted for publication.

[36] Zhuang, F.; Qi, Z.; Duan, K.; Xi, D.; Zhu, Y.; Zhu, H.; Xiong, H.; He, Q. (2020). *A comprehensive survey on transfer learning*, Proceedings of the IEEE, 109(1), 43–76, 2020.

Journal's webpage: http://univagora.ro/jour/index.php/ijccc/

This journal is a member of, and subscribes to the principles of,
the Committee on Publication Ethics (COPE).
https://publicationethics.org/members/international-journal-computers-communications-and-control

Cite this paper as:

T. Sun, D. Zhu, L. He, D. Zhang. (2025). Optimizing Symbolic Execution Path Exploration with a Transfer Learning-Based Strategy, *International Journal of Computers Communications & Control*, 20(5), 6885, 2025.
https://doi.org/10.15837/ijccc.2025.5.6885