Abstract—The code intelligence (CI) models are often black-box and do not offer any insights on the input features that they learn for making correct predictions. This opacity may lead to distrust in their prediction and hamper their wider adoption in safety-critical applications. In recent, the program reduction technique is widely being used to identify key input features in order to explain the prediction of CI models. The approach removes irrelevant parts from an input program and keeps the minimal snippets that a CI model needs to maintain its prediction. However, the state-of-the-art approaches mainly use a syntax-unaware program reduction technique that does not follow the syntax of programs, which adds significant overhead to the reduction of input programs and explainability of models.

In this paper, we apply a syntax-guided program reduction technique that follows the syntax of input programs during reduction. Our experiments on multiple models across different types of input programs show that the syntax-guided program reduction technique significantly outperforms the syntax-unaware program reduction technique in reducing the size of input programs. Extracting key input features from reduced programs reveals that the syntax-guided reduced programs contain more label-specific key input features and are more vulnerable to adversarial transformation when renaming the key tokens in programs. These label-specific key input features may help to understand the reasoning of models’ prediction from different perspectives and increase the trustworthiness to correct classification given by CI models.

Index Terms—Key Input Features, Reduction, Interpretability.

I. INTRODUCTION

Deep neural networks have exaggerated their capability to reduce the burden of feature engineering in complex domains, including code intelligence [1, 2]. While the performance of neural models for intelligent code analysis continues to improve, our understanding of what relevant features they learn for correct classification is largely unknown. Therefore, in this paper, we aim to extract key input features that code intelligence models learn for the target label.

A neural code intelligence (CI) model is a deep neural network that takes a program as input and predicts certain properties of that program as output, for example, predicting method name [3], variable name [4], or type [5] from a program body. Recent studies have shown that state-of-the-art CI models do not always generalize to other experiments [6,7], heavily rely on specific tokens [8,9,10] or structures [11], can learn noisy data points [12,13], and are often vulnerable to semantic-preserving adversarial examples [14,15]. Therefore, it is important to know what input features those CI models learn for making correct predictions. The lack of understanding would hinder the trustworthiness to correct classification given by CI models. Hence, researchers are interested to extract relevant input features that CI models learn for the target label. Such transparency about learned input features is key for wider adoption and application in critical settings such as vulnerability detection or auto-fix suggestion.

Models usually represent an input program as continuous distributed vectors that are computed after training on a large volume of programs. From that, understanding what input features a black-box model has learned is very challenging. For example, code2vec model [16] learns to represent an input program as a single fixed-length high dimensional embeddings, however, the meaning or characteristics of each dimension are not defined. An attention-based approach can be used to enhance important code elements in a program. For example, Bui et al. [17] identify relevant code elements by perturbing statements of the program and combining corresponded attention and confidence scores. However, the attention-based approach poorly correlates with key elements and suffers from a lack of explainability. Recent studies [9,10,18] show that the reduction-based approach can extract relevant input features in programs with offering a better explainability.

Several works have already been conducted by researchers for finding relevant input features in models’ inference. Allamanis et al. [3] use a set of hard-coded features from source code and show that extracting relevant features is essential for learning effective code context. Rabin et al. [19] attempt to find key input features of a label by manually inspecting some input programs of that label. However, the manual inspection cannot be applied to a large dataset due to the vast number of target labels. Suneja et al. [10] and Rabin et al. [9] apply a syntax-unaware program reduction technique, Delta-Debugging [20], to reduce the size of input programs in order to find the minimal snippet that a model needs to maintain its prediction. However, this approach creates a large number of invalid and unnatural programs as it does not follow the syntax of programs during the reduction, which adds significant overhead to the explainability of models. While state-of-the-art approaches use a manual inspection or syntax-unaware program reduction technique, we focus on applying the syntax-guided program reduction technique. In particular, we adopt PERSES [21], a syntax-guided program reduction
In this paper, we apply a syntax-guided reduction technique, rather than syntax-unaware reduction technique, to remove irrelevant parts from an input program and keep the minimal snippet that the CI model needs to maintain its prediction. Given a model and an input program, our approach adopts PERSES \[21\], a syntax-guided reduction technique, to reduce the size of an input program. The approach continues reducing the input program as long as the model maintains the same prediction on the reduced program as on the original program. The main insight is that, by reducing some input programs of a label, we may better extract key input features of that target label. As the syntax-guided technique follows the syntax of input programs, it will always generate valid input programs. Therefore, the approach is more likely to reach the minimal snippet in a smaller number of reduction steps, which will decrease the total reduction time. Moreover, following a syntax-guided technique, the approach can reveal more realistic key input features for the target label. However, for supporting a programming language data, the syntax-guided technique needs to leverage knowledge about program syntax for avoiding generating syntactically invalid programs.

An experiment with two CI models and four types of input programs reveals that the syntax-guided PERSES performs very well compared to the syntax-unaware Delta-Debugging. While PERSES can generate 100% valid programs, Delta-Debugging generates around 10% valid programs only. On average, PERSES removes 20% more tokens, takes 70% fewer reduction steps, and spends 2x less reduction time than Delta-Debugging for reducing an input program. Furthermore, our results show that we can find label-specific key input features by reducing input programs using PERSES, which can provide additional explanation for a prediction and highlight the importance of key input features in programs by triggering 10% more misprediction with 50% fewer adversarial examples.

Contributions. This paper makes the following contributions.

- We apply a syntax-guided program reduction technique for reducing an input program while preserving the same prediction of the CI model.
- We provide a systematic comparison between the syntax-guided program reduction and the syntax-unaware program reduction techniques.
- Our results suggest that the syntax-guided program reduction technique significantly outperforms the syntax-unaware program reduction technique.
- We highlight key input features that CI models learn for the target label using syntax-guided program reduction.
- We show that different program reduction techniques may provide additional explanations for a specific prediction.

II. RELATED WORK

There has been some work in the area of code intelligence that focuses on the understanding of what relevant features a black-box model learns for correct predictions. While some work \[6, 8, 9, 10, 14, 15, 22\] studies the reliance of models on specific features, many works \[3, 9, 10, 17, 18, 19\] focus on finding relevant features for explaining models’ prediction.

A. Learning Representation of Source Code

An input program is usually represented as vector embeddings for processing and analyzing by neural models. Allamanis et al. \[23\] introduced a framework that processed token sequences and abstract syntax trees of code to represent the raw programs. Alon et al. \[16\] proposed an attention-based neural model that uses a bag of path-context from abstract syntax tree for representing any arbitrary code snippets. Allamanis et al. \[4\] constructed data and control flow graphs from programs to encode a code snippet. Hellendoorn et al. \[5\] proposed an RNN-based model using sequence-to-sequence type annotations for type suggestion. There are some surveys on the taxonomy of models that exploit the source code analysis \[11, 12\]. Chen and Monperrus \[24\] also provide a survey that includes the usage of code embeddings based on different granularities of programs. However, these models are often black-box and do not provide any insight on the meaning or characteristic of learned embeddings. What features or patterns these embeddings represent are largely unknown. In this work, we extract key input features that a model learns for predicting a target label as an explanation of learned embeddings.

B. Reliance on Specific Features

Models often learn irrelevant features, simple shortcuts, or even noise for achieving target performance. Compton et al. \[8\] show that the code2vec embeddings highly rely on variable names and cannot embed an entire class rather than an individual method. They investigate the effect of obfuscation on improving code2vec embeddings that better preserves code semantics. They retrain the code2vec model with obfuscated variables to forcing it on the structure of code rather than variable names and aggregate the embeddings of all methods from a class. Following the generalizability of word embeddings, Kang et al. \[6\] assess the generalizability of code embeddings in various software engineering tasks and demonstrate that the learned embeddings by code2vec do not always generalize to other tasks beyond the example task it has been trained for. Rabin et al. \[15\] and Yefet et al. \[14\] demonstrate that the models of code often suffer from a lack of robustness and be vulnerable to adversarial examples. They mainly introduce small perturbations in code for generating adversarial examples that do not change any semantics and find that the simple renaming, adding or removing tokens changes model’s predictions. Sunjea et al. \[10\] uncover the model’s reliance on incorrect signals by checking whether the vulnerability in the original code is missing in the reduced minimal snippet. They find that model captures noises instead of actual signals from the dataset for achieving high predictions. Rabin et al. \[9\] demonstrates that models often use just a few simple syntactic shortcuts for making prediction. Rabin et al. \[13\] also show that models can fit noisy training data with excessive parameter capacity. As models often learn noise or irrelevant features for achieving high prediction performance, the lack...
of understanding of what input features models learn would hinder the trustworthiness to correct classification. Such opacity is substantially more problematic in critical applications such as vulnerability detection or auto-fix suggestion. In this work, we extract key input features for CI models in order to provide better transparency and explaining the predictions.

C. Extracting Relevant Input Features

Several kinds of research have been done in finding relevant input features for models of source code. Allamanis et al. [3] exhibit that extracting relevant features is essential for learning effective code context. They use a set of hard-coded features from source code that integrate non-local information beyond local information and train a neural probabilistic language model for automatically suggesting names. However, extracting hard-coded features from source code may not be available for arbitrary code snippets and in dynamically typed languages. Bui et al. [17] propose a code perturbation approach for interpreting attention-based models of source code. It measures the importance of a statement in code by deleting it from the original code and analyzing the effect on predicted outputs. However, the attention-based approach often poorly correlates with key elements and suffers from a lack of explainability. Rabin et al. [19] attempt to find key input features of a label by manually inspecting some input programs of that label. They extract handcrafted features for each label and train simple binary SVM classification models that achieves highly comparable results to the higher dimensional code2vec embeddings for the method naming task. However, the manual inspection cannot be applied to a large dataset. Wang et al. [18] propose a mutate-reduce approach to find key features in the code summarization models. Suneja et al. [10] and Rabin et al. [9] apply a syntax-unaware program reduction technique, Delta Debugging [20], to find minimal snippet which a model needs to maintain its prediction. By removing irrelevant parts to a prediction from the input programs, the authors aim to better understand important features in the model inference. However, the syntax-unaware approach creates a large number of invalid and unnatural programs during the reduction as it does not follow the syntax of programs, thus increases the total steps and time of reduction. In this work, we apply a syntax-guided program reduction technique that overcomes the overhead raised by the syntax-unaware technique.

III. DESIGN AND IMPLEMENTATION

This section describes our approach of extracting input features for code intelligence (CI) models by syntax-guided program reduction. We use PERSES [21] as the syntax-guided program reduction technique in our study. We first provide an overview of how PERSES works and then describe how we adopt it in the workflow of our approach.

PERSES. Sun et al. [21] have proposed the framework for syntax-guided program reduction called PERSES. Given an input program, the grammar of that programming language, and the output criteria, PERSES reduces the input program with respect to the grammar while preserving the output criteria. It mainly follows the below steps.

- It first parses the input program into a parse tree by normalizing the definition of grammar.
- Then it traverses the tree and determines whether a tree node is deletable (such as follows the grammar and preserves the output criteria). If yes, it prunes the sub-tree from that node and generates a valid reduced program, else it ignores that node and avoids generating invalid programs. Thus, in each iteration of reduction, it ensures generating syntactically valid program variants that preserves the same output criteria.
- Next, the deletion of one node may enable the deletion of another node. Therefore, PERSES is repeatedly applied to the reduced program until no more tree nodes can be removed, which is known as fixpoint mode reduction.
- The final reduced program is called 1-tree-minimal, and any further attempts to reduce the program would generate an invalid program or change the output criteria.

We integrate the PERSES as a black-box framework in our approach for extracting input features of CI models.

Workflow. Figure 1 depicts a high-level view of the workflow in our proposed methodology. Given a set of input programs, our approach reduces each input program using PERSES while preserving the same prediction by the CI model. The approach removes irrelevant parts from an input program and keeps the minimal code snippet that the CI model needs to maintain its prediction. The main insight is that, by reducing some input programs of a target label, we can identify key input features of the CI model for that target label. Our approach follows the below steps.

![Fig. 1: Workflow of our approach.](image-url)
Given an input program $P$ and a CI model $M$, our approach first record the prediction $y$ (i.e. predicted method name) given by the CI model $M$ on the input program $P$, such as $y = M(P)$.

Using PERSSES, we then generate a candidate reduced program $R'$ by removing some nodes from the tree of the input program $P$, such as $R' = \text{PERSSES}(P)$.

If the candidate reduced program $R'$ does not hold the same prediction $y$ by the CI model $M$ (i.e. $y \neq M(R')$), we reject this candidate program and create another candidate program by removing some other nodes from the tree of the input program.

If the candidate reduced program $R'$ preserves the same prediction $y$ by the CI model $M$ (i.e. $y = M(R')$), we continue reduction and iteratively search for the final reduced program $R$ that produces the same prediction $y$.

The final reduced program is 1-tree-minimal, which contains the key input features that the CI model must need for making the correct prediction $y$.

After reducing a set of input programs of a target label, we extract the node type and token value from the abstract syntax tree (AST) of each reduced program. Every extracted element from reduced programs is considered as a candidate input feature. The most common elements are identified as label-specific key input features and other uncommon elements are identified as input-specific sparse features.

**Implementation.** Our approach is model-agnostic and can be applied for various tasks and programming datasets. In this paper, for experimentation of our approach, we study two well-known code intelligence models (CODE2VEC and CODE2SEQ), a popular code intelligence task (METHODNAME) and one commonly used programming language dataset (JAVA-LARGE) with different types of input programs. This section outlines all of these.

1) Task: We use the method name prediction (METHODNAME) task in this study. This task is commonly used by researchers in the code intelligence domain for various applications such as code summarization \[3,25\], representation learning \[16,26\], neural testing \[6,14,15\], feature extraction \[9,18\], and so on \[11,2\]. In the METHODNAME task, a model attempts to predict the name of a method from its body. Figure 2 shows an example of METHODNAME task, where given the following code snippet: "void f(int a, int b) {int temp = a; a = b; b = temp;}", the CODE2VEC model correctly predicts the method's name as "swap".

2) Models: We use the CODE2VEC \[16\] and CODE2SEQ \[26\] code intelligence models for METHODNAME task. Both models use paths from abstract syntax trees (AST) to encode a program. Given a sample expression "a = b;", an example of path context in AST is "a, <NameExpr \[AssignExpr \downarrow IntegerLiteralExpr\], b>".

- **CODE2VEC.** This model extracts a bag of path-context from the AST of the program where each path-context includes a pair of terminal nodes and the corresponding path between them. The model learns embeddings of these path-contexts during training and uses an attention mechanism to aggregate multiple path-contexts to a single code vector. The code vector is used as a representation of the program for making a prediction.
- **CODE2SEQ.** This model also extracts a bag of path-context from the AST of the program but it sub-tokenized each path-context. The CODE2SEQ model uses a bidirectional LSTM to encode paths node-by-node, and another LSTM to decode a target sequence one-by-one.

3) Dataset: For the METHODNAME task, we use the JAVA-LARGE dataset \[26\]. This dataset contains a total of 9,500 Java projects from GitHub, where 9,000 projects are for the training set, 200 projects for the validation set, and 300 projects for the test set. Using training set and validation set, we train both the CODE2VEC and CODE2SEQ models.

4) Input Types: The dataset from GitHub is often imbalanced and contains different sizes and frequencies of input programs. Therefore, we choose different types of input programs from the JAVA-LARGE test set to evaluate the effectiveness of our approach in terms of reduction and feature extraction.

- **Frequent Methods:** We randomly sample a total of 100 input programs from the most occurring method names.
- **Rare Methods:** We randomly sample a total of 100 input programs from the least occurring method names.
- **Smaller Methods:** We randomly sample a total of 100 input programs that contains less than 10 lines of code.
- **Larger Methods:** We randomly sample a total of 50 input programs that has around 100 lines of code.

Moreover, to demonstrate the label-specific key input features, we select correctly predicted input programs from the ten most frequent labels of the JAVA-LARGE test set for feature extraction. Those labels (methods) are: equals, main, setUp, onCreate, toString, run, hashCode, init, execute, and get.

5) Syntax-unaware Reduction Technique: We use the Delta-Debugging algorithm as the syntax-unaware program reduction technique in this study. Zeller and Hildebrandt \[20\] have proposed the Delta-Debugging algorithm to reduce the size of an input program. The algorithm iteratively splits an input program into multiple candidate programs by removing parts of the input program. The algorithm then checks if any resulting candidate program preserves the prediction of the model on the original input program. When the algorithm finds
a candidate satisfying the property, it uses the candidate as the new base to be reduced further. Otherwise, the algorithm increases the granularity for splitting, until it determines that the input program cannot be reduced further.

- **DD-Token**: In the token level approach, Delta-Debugging reduces the size of an input program token by token. We mostly use the **DD-Token** as the default baseline for Delta-Debugging in this study.
- **DD-Char**: In the char level approach, Delta-Debugging reduces the size of an input program char by char. We use the **DD-Char** approach to provide an additional explanation in Section IV-C and Figure 4.

Rabin et al. [9] described more detail on how the Delta-Debugging technique is adopted in the workflow of reducing input programs for CI models.

### IV. RESULTS

In this section, we present the average result of our experiments on the CODE2VEC and CODE2SEQ models and the JAVA-LARGE dataset for different input types.

#### A. Comparative Analysis

Here, we provide a systematic comparison between the syntax-guided program reduction technique and the syntax-unaware program reduction technique. In particular, we compare the syntax-guided PERSES and the syntax-unaware Delta-Debugging in terms of token reduction, valid candidates, reduction steps and reduction time.

1) **Token Reduction**: The goal of PERSES and Delta-Debugging is to remove irrelevant tokens from an input program as much as possible while preserving the same prediction of the CI model. Figure 3a shows their such ability in reducing the size of the original input programs for different input types. We can see that, for all input types, PERSES reduces more tokens from an input program than Delta-Debugging. On average, PERSES removes 20\% more tokens from an input program than Delta-Debugging. The difference is most significant (around 30\%) in LARGE input types and less significant (around 5\%) in RARE input types. This result suggests that PERSES is more powerful than Delta-Debugging in reducing the size of an input program.
2) **Valid Candidates:** In each reduction step, PERSES and Delta-Debugging create a candidate program after removing some irrelevant tokens from an input program, and continue for further reduction. Figure 3b shows their effectiveness in generating valid candidate programs during reduction. For all input types, PERSES always reduces to a valid candidate program (thus, 100% valid candidates) as it follows the syntax of programs during reduction. However, in most cases, Delta-Debugging reduces to an invalid candidate program (only around 10% valid candidates) as it does not follow the syntax of programs. Therefore, after each invalid step, Delta-Debugging backtracks to the previous step and generates another candidate program by removing tokens from some other parts of the program, which increases the overhead in total reduction steps and reduction time.

3) **Reduction Steps:** The reduction is applied repeatedly to an input program until finding the final minimal program, from where no more tokens can be removed. From Figure 3c we can see that PERSES on average can reach the final minimal program within 5 reduction steps. However, Delta-Debugging makes around 20 reductions in FREQUENT-RARE-SMALL input types and more than 50 reductions in LARGE input type, to reach the final minimal program. The Delta-Debugging reduces an input program by a sequence of tokens where PERSES can prune an entire sub-tree from AST. Thus, PERSES takes a much lower number of reduction steps than Delta-Debugging to reach the final minimal program.

4) **Reduction Time:** We now compare the average time taken by PERSES and Delta-Debugging for reducing an input program. As Delta-Debugging takes excessive invalid steps, PERSES is expected to spend less time for program reduction. Figure 3d shows that for all input types, PERSES takes an input program faster than Delta-Debugging, specially in LARGE input type. Where PERSES spend only 2 minutes to reduce an input program and comparatively PERSES takes 30 seconds less time than Delta-Debugging. In LARGE input types, Delta-Debugging spends around 17 minutes for the reduction of a large program but PERSES takes only 8 minutes, which is around 50% less than Delta-Debugging.

**Observation 1:** PERSES allows more token removal than Delta-Debugging and always creates valid candidate programs. Compared to Delta-Debugging, PERSES is more likely to reach the final minimal program in a smaller number of reduction steps, which decreases the total reduction time.

### B. Label-Specific Key Input Features

Here, we provide the summary of extracted input features that CI models learn for predicting the target method name. In our experiment, we consider all tokens in reduced programs as candidate tokens. A label-specific key input feature is a candidate token that appears in at least 50% of reduced programs, where other infrequent tokens are input-specific sparse features. For brevity and page limit, we only show the Top-10 most frequent methods in Table I and Table II.

From Table I considering both CODE2VEC and CODE2SEQ models, we can see that both PERSES and Delta-Debugging identify around 50 tokens, in total, as label-specific key input features in Top-10 methods. However, Delta-Debugging contains a total of 324 candidate tokens in reduced programs, which is 1.36x time higher than PERSES that contains a total of 238 candidate tokens. In some methods, i.e. ‘equals’ and ‘setUp’, the total number of candidate tokens in Delta-Debugging reduced programs is almost 2x time higher than the candidate tokens in PERSES reduced programs. This shows that the tokens found from the reduced programs of Delta-Debugging are more input-specific while the tokens found from the reduced programs of PERSES are more label-specific.

Furthermore, Table II shows the label-specific key input features (sorted by their frequency) extracted by Delta-Debugging and PERSES from its reduced programs. These label-specific key input features can help to understand the prediction of the CI model for a target label. For example, Delta-Debugging and PERSES reveal that “void, args, String, Exception” are key features for the ‘main’ method. It highlights that a sample input program containing those tokens is more likely to be predicted as the ‘main’ method.

### Table I: Summary of key input features in Top-10 methods.

| Method  | Model   | Delta-Debugging (DD) | PERSES |
|---------|---------|----------------------|--------|
|         | Candidate | Key | Candidate | Key |
| equals  | CODE2VEC | 30  | 8   | 12  | 5   |
|         | CODE2SEQ | 28  | 8   | 10  | 5   |
| main    | CODE2VEC | 27  | 5   | 21  | 5   |
|         | CODE2SEQ | 27  | 5   | 4   | 3   |
|         | (both)   | 28  | 5   | 21  | 5   |
| setUp   | CODE2VEC | 36  | 4   | 13  | 5   |
|         | CODE2SEQ | 27  | 4   | 13  | 1   |
|         | (both)   | 41  | 4   | 19  | 5   |
| onCreate | CODE2VEC | 31  | 5   | 20  | 4   |
|         | CODE2SEQ | 24  | 4   | 14  | 3   |
|         | (both)   | 34  | 5   | 24  | 4   |
| toString| CODE2VEC | 21  | 4   | 18  | 5   |
|         | CODE2SEQ | 22  | 3   | 18  | 2   |
|         | (both)   | 23  | 4   | 25  | 5   |
| run     | CODE2VEC | 30  | 5   | 22  | 5   |
|         | CODE2SEQ | 31  | 6   | 13  | 2   |
|         | (both)   | 36  | 6   | 27  | 5   |
| hashCode| CODE2VEC | 13  | 5   | 6   | 5   |
|         | CODE2SEQ | 14  | 5   | 12  | 4   |
|         | (both)   | 15  | 5   | 13  | 5   |
| init    | CODE2VEC | 30  | 3   | 29  | 3   |
|         | CODE2SEQ | 30  | 3   | 10  | 1   |
|         | (both)   | 35  | 3   | 33  | 3   |
| execute | CODE2VEC | 20  | 5   | 14  | 5   |
|         | CODE2SEQ | 16  | 3   | 7   | 3   |
|         | (both)   | 24  | 6   | 14  | 5   |
| get     | CODE2VEC | 36  | 6   | 49  | 4   |
|         | CODE2SEQ | 23  | 0   | 16  | 0   |
|         | (both)   | 58  | 6   | 30  | 4   |
| Top-10  | (both)   | 324 | 52  | 238 | 47  |
TABLE II: Label-specific key input features in Top-10 methods.

| Method | Model | Reduction | Key Input Features |
|--------|-------|-----------|-------------------|
| equals | Code2Vec | DD | if, boolean, Object, o, obj, other, instanceof, Stock |
|        | Code2Seq | DD | if, boolean, Object, obj, other, o, instanceof, Stock |
| main   | Code2Vec | DD | args, void, String, Exception, throws |
|        | Code2Seq | DD | args, void, String, Exception, throws |
| setUp  | Code2Vec | DD | void, throws, Exception, setUp |
|        | Code2Seq | DD | void, throws, Exception, super, setUp |
| onCreate | Code2Vec | DD | void, savedInstanceState, Bundle, onCreate, if |
|        | Code2Seq | DD | void, savedInstanceState, super, onCreate |
| toString | Code2Vec | DD | void, String, arg |
| run    | Code2Vec | DD | void, try, catch, 0, x |
|        | Code2Seq | DD | void, try, catch, 0, Override, x |
| init   | Code2Vec | DD | void, Override |
|        | Code2Seq | DD | void, hashCode, 0, result, null |
| hashCode | Code2Vec | DD | void, Override, result, final, prime |
| execute | Code2Vec | DD | void, throws, BuildException, execute, context |
|        | Code2Seq | DD | void, throws, BuildException, super, execute |
| get    | Code2Vec | DD | if, l, null, return, key, Object |
|        | Code2Seq | DD | if, l, null, return, key, Object |

(a) A sample input program:
```java
public static void f(String[] args) {
    System.setProperty(        
        Constants.DUBBO_PROPERTIES_KEY,        
        "conf/dubbo.properties"
    );
    Main.main(args);
}
```
(b) DD-Char reduced program:
```java
d f(Sg[r]{y(C,\"\")};Main(ar));
```
(c) DD-Token reduced program:
```java
void f(String[ ]args)((\"");(args));
```
(d) PERSES reduced program:
```java
void f(String args) { }
```

Observation 3: Different program reduction techniques may provide additional explanations for better transparency of a specific prediction.

D. Key Targeted Adversarial Attacks on Models

Here, we highlight the importance of key input features in programs by evaluating the adversarial generalizability [15] or robustness [14] of CI models in terms of the extracted key input features. We generate adversarial examples by applying semantic-preserving variable renaming transformation on programs, similar to [15], where we separately change each variable and all of its occurrences in the program with token var. We particularly compare the prediction of CI models before and after the variable renaming. In this experiment, we generate three types of adversarial sets: actual set, key set, and reduced set. First, in actual set, we target the actual initial programs and generate candidate transformed programs by considering all variables. Second, in key set, we also target the actual initial programs but generate candidate transformed programs by considering variables that occur in the key feature list. Third, in reduced set, we directly target the reduced programs for generating candidate transformed programs. The results of change in prediction (misprediction) for variable renaming transformation are shown in Table III.

According to Table III, on average, the number of generated candidate transformed programs from the actual set are around 3x times higher than the initial programs, however, only 12% of them trigger misprediction. Next, the number of generated candidate transformed programs from the key set are around 1.5x times higher than the initial programs and trigger 22% misprediction. Although the key adversarial set

**Method by CI models.**

Observation 2: PERSES reveals more label-specific key input features in its syntax-guided reduced programs, while Delta-Debugging contains more input-specific sparse features in its syntax-unaware reduced programs.

C. Multiple Explanation for a Specific Prediction

Different program simplification approaches, i.e., Delta-Debugging and PERSES, provide a different set of key features for a target label by a CI model (Table I). Those different features can help us to find multiple explanations for a specific prediction. For instance, the CODE2SEQ predicts the code snippet in Figure 4a as the main method. The Delta-Debugging with char-based program reduction (DD-Char) gives the minimal program in Figure 4b, that CODE2SEQ can predict as main. We can see the presence of the Main identifier in the method body of Figure 4a, which is one of the important tokens for the target prediction. On the other hand, the Delta-Debugging with token-based program reduction (DD-Token) gives the minimal program in Figure 4a, which suggests the argument args has an important role in the target prediction. However, with the AST-based program reduction
### Table III: Adversarial evaluation with key features.

| Reduction | Model | Adversarial Set | #Initial | #Transformed | Misprediction |
|-----------|-------|-----------------|----------|-------------|---------------|
|           |       |                 |          |             |               |
| Delta-Debugging | CODE2VEC | Actual Sample | 328      | 1148        | 135           | 11.76         |
|           |       | Key Token       | 328      | 722         | 107           | 14.82         |
|           |       | Reduced Sample  | 328      | 379         | 117           | 30.87         |
| PERSES    | CODE2VEC | Actual Sample | 287      | 836         | 109           | 13.04         |
|           |       | Key Token       | 287      | 530         | 102           | 19.25         |
|           |       | Reduced Sample  | 287      | 267         | 134           | 50.19         |
|           | CODE2SEQ | Actual Sample | 280      | 658         | 93            | 14.13         |
|           |       | Key Token       | 280      | 211         | 80            | 37.91         |
|           |       | Reduced Sample  | 280      | 160         | 104           | 65.00         |

contains 50% less candidate transformed programs than the actual adversarial set, they trigger 10% more misprediction. On the other hand, the reduced programs are the minimal program that CI models keep for preserving their target prediction. Therefore, the number of generated candidates transformed programs from the reduced set are the lowest as there are fewer tokens to apply transformations. However, the transformation on reduced programs is more powerful and triggers the highest percentage of misprediction. Moreover, comparing between Delta-Debugging and PERSES, in most cases, PERSES generated candidates transformed programs shows a higher rate of misprediction than Delta-Debugging.

**Observation 4:** The adversarial programs based on key input features trigger 10% more misprediction with 50% fewer candidates. The PERSES generated candidate programs are more vulnerable to adversarial transformation than Delta-Debugging, thus, highlighting the importance of key input features in programs.

## V. Threats to Validity and Future Plan

**Evaluation.** We evaluated our approach for METHOD_NAME task with two CI models, four input types of randomly selected input programs, and Top-10 most frequent method names. Despite our best effort, it is possible that experiments with different models, tasks, and datasets may produce different results. Our further plan includes a detailed study with a variety of models, tasks, and larger datasets.

**Challenges.** One challenge we have for running PERSES is that it loads the model in each reduction step while Delta-Debugging loads the model once at the beginning of reduction. For a fair comparison between them, we only consider the program reduction time and ignore the model loading time. We are working on optimizing the model loading time for PERSES.

Another challenge for running Delta-Debugging, when there are multiple subsets that hold the same target criteria, Delta-Debugging sometimes gets stuck at that point. To keep the reduction process working, we temporarily used a timer to kill the current step and jump to the next step.

**Artifacts.** We will publicly share the artifacts of this study at https://github.com/mdrafiqulrabin/rm-dd-perses.

## VI. Conclusion

In this paper, we apply the syntax-guided program reduction technique, PERSES, for reducing an input program while preserving the same prediction of the CI model. The goal is to extract label-specific key input features of target labels for CI models from syntax-guided reduced programs. We evaluate PERSES on two popular CI models across four types of input programs for the method name prediction task. Our results suggest that the syntax-guided program reduction technique (PERSES) significantly outperforms the syntax-unaware program reduction technique (Delta-Debugging) in reducing different input programs. Moreover, we extract key input features that CI models learn for a target label, by reducing some input programs of that label using PERSES. The result shows that PERSES mostly keeps label-specific key input features in its syntax-guided reduced programs than in Delta-Debugging’s syntax-unaware reduced programs. We also observe that the syntax-guided candidate programs are more vulnerable to adversarial transformation when renaming the key tokens in programs. By identifying those key input features, we can better understand the learned behaviors of CI models from multiple explanations, which may improve the trustworthiness of models to correct prediction.

**Acknowledgement**

This study has been done as coursework in the Department of Computer Science at the University of Houston (Course: COSC 6321 - Research Methods in Computer Science; Instructor: Omprakash D Gnawali). We organized an in-class conference (Research Quest 2021) and submitted our poster/papers as homework to the conference.

**References**

[1] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, “A survey of machine learning for big code and naturalness,” in *ACM Computing Surveys*, vol. 51, no. 4. New York, NY, USA: Association for Computing
[10] S. Suneja, Y. Zheng, Y. Zhuang, J. A. Laredo, and A. Morari, “Probing model signal-awareness via prediction-preserving input minimization,” in Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 945–955. [Online]. Available: https://doi.org/10.1145/3468264.3468545

[11] M. R. I. Rabin and M. A. Alipour, “Code2snapshot: Using code snapshots for learning representations of source code,” 2021. [Online]. Available: https://arxiv.org/abs/2111.01097

[12] M. Allamanis, “The adverse effects of code duplication in machine learning models of code,” in Proceedings of the ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, ser. Onward! 2019. ACM New York, NY, USA, 2019, pp. 143–153. [Online]. Available: https://doi.org/10.1145/3359591.3359735

[13] M. R. I. Rabin, A. Hussain, V. J. Hellendoorn, and M. A. Alipour, “Memorization and generalization in neural code intelligence models,” 2021. [Online]. Available: https://arxiv.org/abs/2106.08704

[14] N. Yefet, U. Alon, and E. Yahav, “Adversarial examples for models of code,” in Proceedings of the ACM on Programming Languages, ser. PACMPL 2020, vol. 4, no. OOPSLA. New York, NY, USA: Association for Computing Machinery, 2020, pp. 162:1–162:30. [Online]. Available: https://doi.org/10.1145/3428230

[15] M. R. I. Rabin, N. D. Bui, K. Wang, Y. Yu, L. Jiang, and M. A. Alipour, “On the generalizability of neural program models with respect to semantic-preserving program transformations,” in Information and Software Technology, vol. 135. Amsterdam, Netherlands: Elsevier, 2021, p. 106552. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584921000379

[16] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, “Code2vec: Learning distributed representations of code,” in Proceedings of the ACM on Programming Languages, ser. PACMPL 2019, vol. 3, no. POPL. New York, NY, USA: Association for Computing Machinery, 2019, pp. 40:1–40:29. [Online]. Available: https://doi.org/10.1145/3290353

[17] N. D. Q. Bui, Y. Yu, and L. Jiang, “Autofocus: Interpreting attention-based neural networks by code perturbation,” in Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering, ser. ASE 2019. New York, NY, USA: Association for Computing Machinery, 2019, pp. 38–41. [Online]. Available: https://doi.org/10.1109/ASE.2019.00014

[18] Y. Wang, F. Gao, and L. Wang, “Demystifying code summarization models,” 2021. [Online]. Available: https://arxiv.org/abs/2102.04625v1

[19] M. R. I. Rabin, A. Mukherjee, O. Gnawali, and M. A. Alipour, “Towards demystifying dimensions of source
code embeddings,” in Proceedings of the 1st ACM SIGSOFT International Workshop on Representation Learning for Software Engineering and Program Languages, ser. RL+SE&PL 2020. New York, NY, USA: Association for Computing Machinery, 2020, pp. 29–38. [Online]. Available: https://doi.org/10.1145/3416506.3423580

[20] A. Zeller and R. Hildebrandt, “Simplifying and isolating failure-inducing input,” in IEEE Transactions on Software Engineering, vol. 28, no. 2. New York, NY, USA: IEEE Press, 2002, pp. 183–200. [Online]. Available: https://doi.org/10.1109/32.988498

[21] C. Sun, Y. Li, Q. Zhang, T. Gu, and Z. Su, “Perses: Syntax-guided program reduction,” in Proceedings of the 40th International Conference on Software Engineering, ser. ICSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, pp. 361–371. [Online]. Available: https://doi.org/10.1145/3180155.3180236

[22] M. R. I. Rabin, K. Wang, and M. A. Alipour, “Testing neural program analyzers,” 34th IEEE/ACM International Conference on Automated Software Engineering (Late Breaking Results-Track), 2019. [Online]. Available: https://arxiv.org/abs/1908.10711

[23] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, “Learning natural coding conventions,” in Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, ser. FSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, pp. 281–293. [Online]. Available: https://doi.org/10.1145/2635868.2635883

[24] Z. Chen and M. Monperrus, “A literature study of embeddings on source code,” 2019. [Online]. Available: https://arxiv.org/abs/1904.03061

[25] M. Allamanis, H. Peng, and C. A. Sutton, “A convolutional attention network for extreme summarization of source code,” in Proceedings of the 33nd International Conference on Machine Learning, ser. ICML 2016. Open Access: Proceedings of Machine Learning Research (PMLR), 2016, pp. 2091–2100. [Online]. Available: http://proceedings.mlr.press/v48/allamanis16.html

[26] U. Alon, O. Levy, and E. Yahav, “code2seq: Generating sequences from structured representations of code,” in International Conference on Learning Representations, ser. ICLR 2019. Open Access: OpenReview.net, 2019. [Online]. Available: https://openreview.net/forum?id=H1gKYo09tX