A Learning-Based Approach to Static Program Slicing

AASHISH YADAVALLY, University of Texas at Dallas, USA
YI LI, University of Texas at Dallas, USA
SHAOHUA WANG, Central University of Finance and Economics, China
TIEN N. NGUYEN, University of Texas at Dallas, USA

Traditional program slicing techniques are crucial for early bug detection and manual/automated debugging of online code snippets. Nevertheless, their inability to handle incomplete code hinders their real-world applicability in such scenarios. To overcome these challenges, we present NS-Slicer, a novel learning-based approach that predicts static program slices for both complete and partial code. Our tool leverages a pre-trained language model to exploit its understanding of fine-grained variable-statement dependencies within source code. With this knowledge, given a variable at a specific location and a statement in a code snippet, NS-Slicer determines whether the statement belongs to the backward slice or forward slice, respectively.

We conducted a series of experiments to evaluate NS-Slicer's performance. On complete code, it predicts the backward and forward slices with an F1-score of 97.41% and 95.82%, respectively, while achieving an overall F1-score of 96.77%. Notably, in 85.20% of the cases, the static program slices predicted by NS-Slicer exactly match entire slices from the oracle. For partial programs, it achieved an F1-score of 96.77%–97.49% for backward slicing, 92.14%–95.40% for forward slicing, and an overall F1-score of 94.66%–96.62%. Furthermore, we demonstrate NS-Slicer's utility in vulnerability detection (VD), integrating its predicted slices into an automated VD tool. In this setup, the tool detected vulnerabilities in Java code with a high F1-score of 73.38%. We also include the analyses studying NS-Slicer's promising performance and limitations, providing insights into its understanding of intrinsic code properties such as variable aliasing, leading to better slicing.

CCS Concepts: • **Computing methodologies → Neural networks**; • **Software and its engineering → Language features**;

Additional Key Words and Phrases: AI4SE, Neural Networks, Static Slicing, Pre-Trained Language Models, Vulnerability Detection, Debugging

ACM Reference Format:
Aashish Yadavally, Yi Li, Shaohua Wang, and Tien N. Nguyen. 2024. A Learning-Based Approach to Static Program Slicing. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 97 (April 2024), 27 pages. https://doi.org/10.1145/3649814

1 INTRODUCTION

Online forums, such as StackOverflow (S/O), are a trove of knowledge for developers, offering excellent resources for those seeking solutions to technical challenges. However, the standard practice of software reuse via copy-and-paste is limited, as the copied code fragment can possess vulnerabilities (i.e., exploitable defects) — potentially posing significant risks to the applications that adopt them. For instance, [Verdi et al. 2022] studied the code snippets extracted from 1,325 S/O

Authors' addresses: Aashish Yadavally, University of Texas at Dallas, Dallas, USA, aashish.yadavally@utdallas.edu; Yi Li, University of Texas at Dallas, Dallas, USA, yi.li@utdallas.edu; Shaohua Wang, Central University of Finance and Economics, Bejing, China, davidshwang@ieee.org; Tien N. Nguyen, University of Texas at Dallas, Dallas, USA, tien.n.nguyen@utdallas.edu.

ACM 2475-1421/2024/4-ART97
https://doi.org/10.1145/3649814

answers and reported 99 vulnerable ones that subsequently migrated to 2,589 GitHub repositories. [Hong et al. 2021] collected 1,958,283 S/O posts tagged with C, C++, and Android. They found 12,458 insecure posts (14,719 insecure code snippets) and confirmed that the latest versions of 151 out of 2,000 popular C/C++ open-source software contain at least one of these insecure S/O code snippets. Another study revealed that 15.4% of 1.3 million Android applications contain security-related S/O code snippets [Fischer et al. 2017]. Ragkhitwetsagul *et al.* [Ragkhitwetsagul et al. 2021] also reported that the code snippets imported from S/O into the Github projects could contain issues including defects, vulnerabilities, performance bottlenecks, copyright issues, *etc.*

Detecting errors and vulnerabilities in S/O code snippets at an early stage offers numerous benefits. First, early detection can help prevent the costs associated with fixing vulnerable code later in the development cycle (say, after deployment or when in production). Second, it promotes proactive risk mitigation while enabling an efficient development workflow. Here, developers can address issues without disrupting the ongoing projects. Consider the alternative of detecting errors/vulnerabilities after adopting the code snippet. If found error-prone, the efforts of integrating the S/O code into existing code repositories would be lost. Furthermore, one cannot identify whether the error arises due to the code snippet, the existing source code, or the combination of both.

Recognizing the need to detect errors early, security researchers have introduced several manual and automated approaches for vulnerability detection (VD). Some of these leverage static and dynamic program analysis tools [Ayewah et al. 2008; Checkmarx 2023], using which, for instance, one can alert developers of potential synchronization issues that might lead to unsafe thread interactions. Nonetheless, these tools are often limited to specific vulnerabilities, e.g., buffer overflow [CWE-120 2023], SQL injection [CWE-89 2023], cross-site scripting [CWE-79 2023], authentication bypass [CWE-290 2023], *etc.* More importantly, since these approaches are program analysis-based, they necessitate that *the code is complete.* Besides, those based on dynamic analysis techniques also require the setting up of dedicated, operational test environments.

Alternatively, some machine and deep learning (ML & DL)-based VD approaches attempt to implicitly learn the vulnerability patterns, relying on program representations such as static program slices (VulDeePecker [Li et al. 2018]), program dependence graphs (VulCNN [Wu et al. 2022]), *etc.* In particular, [Li et al. 2018] draw inspiration from how developers manually attempt to locate such errors, often narrowing down the source code to its essential elements by constructing the static program slices (i.e., backward/forward) for the variables. In this context, the *backward slice* of a variable at a program location consists of the statements that can impact the value of that variable, and the *forward slice* includes statements that might be affected by changes to the variable.

Program slicing is also useful in both manual and automated debugging. [Francel and Rugaber 2001] reported that slicing-aided debugging helps the developer focus on the minimal subprogram containing program faults, improving code understanding, and ability to fix program faults. For instance, consider the C++ program in Fig. 1, which inputs a list of integers interactively and: (1) counts the number of non-negative integers in the list, (2) finds the maximum and minimum negative integers in the list. Here, once the developer determines that the output value for exactly one value is incorrect (i.e., `currentMaximum` on line 35), the following can be omitted from the fault area: lines *5, 6, 19, 21, 24, 25, 29, 30, 33, 35* – thus decreasing the fault area from the original by 28%.

The state-of-the-art approaches for static program slicing, however, require access to the source code as complete program units, at the very least, at the method-level granularity. As a result, their applicability in detecting vulnerabilities in S/O code snippets is hindered. First, the program analysis (PA)-based approaches, such as JavaSlicer [Galindo et al. 2022], rely on the construction of a system dependence graph (SDG), which is only feasible for complete code. In the case of incomplete code, they fail due to the presence of undeclared variables and unknown types. The next alternative could be to build a more local representation such as the PDG and extract the slices by conducting a

```
1  #include<iostream:h>
2  int  main() {
3    int  currentValue;
4    int  currentMaximum;
5    int  currentMinimum;                                              ...
19   currentMinimum = 0;
20   currentMaximum = 0;
21   nonNegativeCount = 0;                                             ...
32   cout  << end1 <<end1 <<end1;
33   cout  <<"The number of non-negative numbers in the  list  was "  << nonNegativeCount << end1;
34   cout  <<  "The maximum negative number in the  list  was "  << currentMaximum << end1;
35   cout  <<  "The minimum negative number in the  list  was "  << currentMinimum << end1;
36 }
```

Fig. 1. A buggy C++ program (fault area: line 20) [Francel and Rugaber 2001]

data-flow analysis. However, due to the missing declared data types, missing variable declarations, the absence of external library imports, *etc.*, most PA tools fail to build PDGs for incomplete code. When wrapped around dummy method signatures, some tools like [Joern 2023] construct PDGs in a best-effort manner, however, at the cost of several *misses*. We conducted a preliminary empirical study analyzing the reasons behind such misses (see Section 3). Finally, another alternative would be to use a deep learning-based PDG-building tool, such as NeuralPDA [Yadavally et al. 2023], which works for incomplete code. However, it only operates at the statement-level and does not provide fine-grained dependencies among the variables, a prerequisite for extracting slices.

In this paper, we present NS-Slicer, a neural network-based static program slicing approach that produces program slices for variables within (in)complete code snippets. Our tool leverages a pre-trained language model (PLM) to capture the interactions between a variable at a specific location and all other statements in the given program, as well as the elements within those statements. With this knowledge, during the training phase, NS-Slicer learns to predict whether a program statement belongs to the backward or forward slice. During inference, the sequential nature of the PLM within NS-Slicer enables it to be applied to both complete and incomplete code.

The rationale behind the use of PLMs for our problem is the nature of their pre-training tasks, such as, masked language modeling (MLM) [Feng et al. 2020], edge prediction and node alignment [Guo et al. 2021], which have shown to help them learn the syntactic structure in code [Hernández López et al. 2023], and the data-flow information [Guo et al. 2021]. Furthermore, there is an inherent difference in the nature of a backward slice and a forward slice, each of which contain the set of statements that either affect, or are affected by a variable at a program location, respectively. To capture this nuance, we incorporate distinct components within the *static slice decoding* phase in NS-Slicer, such that, for a variable at a given program location, they predict whether a program statement belongs to the set of statements in the backward or forward slice, respectively. Let us use the term **variable-statement dependency** to refer such dependency between a variable at a program location and a statement in a static slice. That is, *a statement is said to have a variable-statement dependency with a variable v at a location* if either that statement has potential influence on the value of v (backward slicing) or the value of the statement has potential to be affected by v (forward slicing). Given that PLMs have demonstrated proficiency in capturing program dependencies at the statement level [Guo et al. 2021; Yadavally et al. 2023], we posit that they possess the ability to learn and predict variable-statement dependencies at a finer token level for static slicing. In fact, our empirical evaluation (Section 6) substantiates this hypothesis, affirming our finding on the PLMs' capacity to capture variable-statement dependencies for static slicing.

In essence, program slicing can be viewed as a semantic knowledge-probing downstream task for the PLMs. Accordingly, we conduct an extensive evaluation – assessing the performance of NS-Slicer on both complete and partial programs. We note that on complete code, our tool predicts the backward and forward slices with an F1-score of 97.41% and 95.82%, respectively, while recording an overall F1-score of 96.77%. Moreover, in 85.20% of the cases, the program slices predicted by NS-Slicer exactly match the ground-truth static program slices, and are executable. In the case of partial programs, it records an F1-score of 96.77%–97.49% for backward slicing, 92.14%–95.40% for forward slicing, and an overall F1-score of 94.66%–96.62%.

NS-Slicer empowers program analysis approaches, which, traditionally rely on static program slicing and require complete code, to operate on partial code. We explore this utility via an extrinsic evaluation of NS-Slicer on vulnerability detection. In this experiment, we plugged our tool within VulDeePecker [Li et al. 2018] such that it makes use of the predicted static program slices to train the model and then detect the presence of vulnerabilities. The motivation behind such a design is that VulDeePecker is unusable for predicting vulnerabilities in partial code due to the failure of its inherent program slicing tool. We observed that our tool achieved an overall F1-score of 73.38%, improving over the baseline by 15.1%. Finally, we conducted a qualitative evaluation of NS-Slicer, aiming to analyze its understanding of intrinsic code properties such as variable aliasing. In this case, we note that the performance of our tool dropped by only 12.31%. We validated this probe with case studies, studying our tool's promising performance and limitations.

In brief, this paper makes the following major contributions:

(1) NS-Slicer is the *first learning-based approach to static program slicing* with a high accuracy, which *extends the applicability of slicing techniques to incomplete code.*
(2) A *new finding that PLMs are capable of capturing the variable-statement dependencies* for the purpose of static program slicing.
(3) We conduct a *comprehensive evaluation*, providing an in-depth analysis of model performance, including ablation studies, qualitative probes, and case studies.
(4) We demonstrate *NS-Slicer's utility* in detecting vulnerabilities in incomplete code.

2 MOTIVATING EXAMPLE

In this section, let us study a real-world example to illustrate the problem and motivate our solution.

2.1 Early Detection of Vulnerabilities in StackOverflow Code Snippets

Fig. 2 illustrates a code snippet that is part of an answer to the question in StackOverflow post #16180130. The goal of this code snippet is to split a file and read the actual bytes from the corresponding file. As reported in the empirical study by [Ragkhitwetsagul et al. 2021], this code snippet was later copied and incorporated into the class `LineRecordReader.java` of the `Hadoop` project.

The program crashed with a Null Pointer Exception (NPE) on line 13 when the variable `in` was `null`, i.e., not pointing to any object, and the method `readLine` was called on `in`. The NPE can occur in two cases in which the variable `in` was not initialized properly: either on line 2 or line 10. Note that both occurrences refer to the parameter `fileIn`. In the case of line 2, `in` could be `null` due to a failed attempt at opening the file associated with the initialization of `fileIn`. In the case of line 10, this could be due to the same failed file opening, or for seeking the incorrect file on lines 7–8. The subsequent migration of this StackOverflow code snippet resulted in the incorrect file seeking vulnerability to be exhibited in the `Hadoop` project. Therefore, *an early detection of vulnerabilities in StackOverflow code snippets, before their adoption into code repositories is desirable.*

```
1   if ( codec != null )
2     in = new LineReader ( codec.createInputStream ( fileIn ), job );
3     end = Long.MAX_VALUE;
4   } else {
5     if ( start != 0 )
6         skipFirstLine = true;
7         –start;
8         fileIn.seek ( start );
9     }
10    in = new LineReader ( fileIn, job );
11  }
12  if ( skipFirstLine )
13    start += in.readLine ( new Text() , 0,
14        ( int ) Math.min ( ( long ) Integer.MAX_VALUE, end – start ) );
15  }
```

Fig. 2. A code snippet in StackOverflow post #16180130

2.2 Program Slicing for Incomplete Code Snippets

To detect vulnerabilities either manually or automatically, it is helpful to focus only on the important statements that can impact the value of any variable of interest, e.g., the variable `in` on line 13. This is referred to as the *slicing criterion*, and the set of all such statements as the *backward slice*. In contrast, the *forward slice* contains the set of statements which might be affected by a change to the slicing criterion. Since we focus on static analysis, the backward slice includes both the NPE-inducing statements, i.e., lines 2 and 10. Moreover, since the related statements are control-dependent on the `if`-statement, the backward slice for variable `in` on line 13 contains lines 12, 10, 8, 7, 6, 5, 2, and 1.

The approaches to *program slicing for incomplete code* are limited. *First*, the traditional program slicing tools, e.g., JavaSlicer [Galindo et al. 2022], leverage a system dependence graph (SDG) to extract program slices. However, this requires access to *complete code along with all of their dependencies*. In Section 3, we attempted to build the program slices for incomplete code snippets from S/O using JavaSlicer. We noticed that it did not produce any slices for code snippets that contain third-party API elements from unknown libraries, or undeclared variables and types.

Second, an alternative solution is to use an automated tool to construct a program dependence graph (PDG) and then perform data/control flow analysis on the PDG to extract the program slice. We conducted an empirical study on applying the state-of-the-art program analysis tool, named Joern [Joern 2023], on incomplete code snippets. We set this up by wrapping the code snippets with method signatures and including additional input parameters when required. However, we noticed that Joern either did not work or produced incomplete PDGs, mainly for the following reasons: (1) missing declared data types, (2) missing variable declarations, (3) certain data types are unresolved due to the missing imports for external libraries, (4) references to undeclared libraries, (5) inability to process templates. We present this study in more detail in Section 3.

Finally, one could also build a PDG using a learning-based approach as in NeuralPDA [Yadavally et al. 2023], which works well for code snippets. By construction, it operates at the statement level, only providing information about inter-statement dependencies. It does not: (1) distinguish between the data and control dependence edges at the fine-grained level, (2) identify the variable upon which the data dependence edge is being predicted. However, the follow-up data/control-flow analysis to extract the program slices from a PDG requires fine-grained variable/argument-specific dependencies. Hence, it is not possible to use NeuralPDA for this purpose.

3 PRELIMINARY EMPIRICAL STUDY

In this section, we present our preliminary empirical study aiming to *qualitatively probe the program dependence graphs constructed by traditional program analysis tools like [Joern 2023] for incomplete code snippets*. The rationale for this study is rooted in the role that program dependence analysis plays in program comprehension as well as facilitating manual/automated debugging.

RQ. *How reliable are the PDGs produced by Joern for incomplete code?*

3.1 Dataset

In this study, we utilized a dataset provided by [Verdi et al. 2022], which consists of 99 incomplete code snippets from StackOverflow spanning 31 different vulnerability types. These snippets were subsequently incorporated into 2,589 GitHub repositories. Our primary objective is to evaluate the effectiveness of Joern in producing the PDGs for these S/O code snippets.

3.2 Procedure

To better understand Joern's efficacy in handling partial code and the conditions resulting in its failure to capture dependencies, we: (a) first, ran Joern on the 99 S/O code snippets (when needed, some instances were wrapped around dummy method signatures); (b) manually inspected the generated CFG/PDGs. We conducted a fine-grained analysis for all 99 instances.

3.3 Empirical Results

Based on our analysis, we grouped Joern outputs for these code snippets into four categories, listing the root causes for its failure in each category [1]:

(1) **Incorrect Outputs**: In 47 cases, Joern either misses or incorrectly predicts multiple control-flow, or data and control-dependence edges.
(2) **Erroneous Instances**: In 30 cases, Joern produces error messages, typically of the form *"Could not find type member. type=XYZ, member=abc."*. Here, XYZ is a type name and *abc* is the corresponding identifier (i.e., the name of a variable or field) in the code snippet. Note that Joern can produce multiple such errors in a single code snippet. Moreover, running Joern on the incomplete code snippets directly without wrapping them with such dummy method signatures increases the number of such erroneous instances to 49.
(3) **Empty CFG, PDG, or both**: In 7 of the cases, Joern does not produce any nodes/edges for either the CFG, PDG, or both.

Broadly, the reasons for the cases in (1)–(3) can be summarized as follows:

- All data dependencies related to a parameter with unknown parameter type are ignored.
- For an unresolved external API class/method/field or an unresolved external data type, dependencies to/from the statements referencing it are ignored.
- All edges related to objects constructed with unresolved/undeclared class are ignored.
- Inaccessible header files leads to undeclared variables, and all dependencies concerning the undeclared variables are missed.
- Missing variable declarations.
- Missing declared data types; unresolved data types from missing external library imports.
- Due to missing class hierarchies, Joern fails to recognize the inherited attributes and skips the corresponding data dependencies.
- Unresolved API references result in Joern skipping edges to/from these statements.
- Joern cannot handle templates, `typedef` declarations, etc.

[1]Refer to our project website for the complete, fine-grained analysis results: https://github.com/aashishyadavally/ns-slicer/

```
1   string subTag(string s, string a, string b){
2     std::string lower_s;
3     std::transform(s.begin(), s.end(), lower_s.begin(), ::tolower);
4     std::transform(a.begin(), a.end(), a.begin(), ::tolower);
5     auto position=lower_s.find(a);
6     while(position!=-1){
7       s.replace(position, a.size(), b);
8       position=lower_s.find(a);
9     }
10    return s;
11  }
```

Fig. 3. A code snippet in StackOverflow post #40577390

(4) **Correct Outputs**: Joern produces correct CFG/PDGs in 15 of the cases, mainly because:
- Some of the instances are complete methods, classes, or files.
- In some cases where the code snippet is incomplete, all the declarations are available, or the variables use primitive data types.
- Some code snippets possess no dependencies as it is just a group of structure definitions (i.e., group of `struct` objects).

3.4 An Illustrating Example

Let us consider the code snippet in Fig. 3 which corresponds to StackOverflow answer ID 40577390 (snippet9 in the dataset). This code snippet was identified as vulnerable when `lower_s` and `s` do not have the same size as being used in the `transform` function. To capture such a vulnerability, it is imperative to be aware of the data dependencies of `s` and `lower_s` from the statements 1 and 2 respectively, to the statement 3. However, since `transform` is an external API, Joern does not recognize it and skips the dependencies for all statements to/from the ones using it. Therefore, the CFG/PDG produced via Joern for this example is not useful for a systematic analysis.

3.5 Conclusion

From Section 3.3, we can see that Joern lacks the ability to construct PDGs, or produces severely under-representative PDGs for incomplete code. In Section 3.4, we highlight the effect such under-representative PDGs can have for the automatic detection of vulnerabilities in incomplete code. Therefore, to establish early bug detection, as well as to facilitate debugging in incomplete code, we observe a need for alternative approaches to derive such dependencies and the subsequent static program slices.

Conclusion. Our findings highlight the inadequacies of traditional program analysis tools in effectively capturing the program dependencies in incomplete code and helping manual and automated debugging.

4 KEY IDEAS

To improve program slicing for incomplete code, we propose NS-Slicer, a neural network-based *partial program slicing framework* that learns to derive the program slices for any criterion (i.e., any variable at any location in an incomplete code snippet). NS-Slicer for (in)complete code can be useful for tasks that can tolerate a low level of errors and imprecision, gaining scale and time efficiency as a trade-off. In designing NS-Slicer, we have the following key ideas:

4.1 [Key Idea 1] Neural Network-Based Partial Program Slicing

Traditional program analysis approaches fail to build program slices for incomplete code (Observation 2.2). Thus, we present an alternative learning-based approach for static program slicing that learns to analyze the fine-grained dependencies among the variables in and across the statements within a complete program in open-source projects (e.g., GitHub) – *enabling its application to predict program slices for (in)complete code snippets*. Our intuition for such a formulation is driven by: (1) traditional program slicing [Galindo et al. 2022] techniques work well for complete code, enabling us to build the training data, i.e., ⟨*Program*, *Slicing Criterion*, *Backward/Forward Slices*⟩ tuples; (2) independence of PLM approaches on the completeness of the program due to their sequence-based nature, i.e., due to taking in a sequence of tokens as input, which can easily be extracted for both complete and incomplete code.

In this work, we aim to imitate the process of building a static program slice from the program dependence graphs. In the traditional approach, for a specific slicing criterion: *first*, one would build the PDG; *second*, apply data-flow analysis to determine which variables influence, or are influenced by the slicing criterion; *third*, leverage the data-flow analysis to traverse the PDG and identify the statements that influence, or are influenced by the slicing criterion, respectively.

4.2 [Key Idea 2] Pre-Trained Language Model for Variable-Statement Dependency Learning

Previous studies [Guo et al. 2021; Hernández López et al. 2023] have demonstrated the ability of PLMs in learning both syntactic and semantic properties in source code. Furthermore, NeuralPDA [Yadavally et al. 2023] reinforces the potential of attention-based models to learn program dependencies at the statement level (i.e., among statements). On this basis, we posit that PLMs can be leveraged to discern the *variable-statement dependencies* for program slicing. Given its data flow-specific learning objectives, we opted for GraphCodeBERT [Guo et al. 2021] as the PLM in NS-Slicer. We expect that GraphCodeBERT, within NS-Slicer, will leverage this knowledge to apply flow analyses across all statements to identify the variable-statement dependencies for slicing, i.e., to identify the statements that influence, or are influenced by the slicing criterion. In fact, we present in Section 6.4.3 a case study to probe and validate such a dependency learning.

4.3 [Key Idea 3] Mimic Program Slice Construction with Dedicated Multi-Pass Classifiers

The PLM in NS-Slicer contextualizes source code tokens to produce syntax and semantics-aware token representations, which also encapsulate the knowledge of the *variables* across statements that either influence, or are influenced by the slicing criterion. By pooling together the token representations of the individual tokens that make up the variable at the slicing criterion and all statements in a given program, we can obtain the corresponding slicing criterion and program statement representations.

Next, we mimic the PDG traversal by designing dedicated multi-pass classifiers, which, for each program statement, utilize the corresponding pooled statement representation to ascertain whether the statement belongs to the backward or forward slice, respectively. Two fundamental factors drive the design of such distinct classifiers. *First*, backward and forward slices are inherently different, as each comprises statements that either affect or are affected by a slicing criterion. We need distinct slicing classifiers to capture this nuance effectively. *Second*, static slicing considers all possible paths to/from the slicing criterion. To probe all such paths, we need to inspect the set of all statements that either precede or follow the specified criterion. However, the cardinality of these statement sets is not constant and is determined by the location of the criterion within the program. Therefore, we need a multi-pass classifier, where each pass involves considering the criterion variable and a statement from the backward/forward statement sets – to determine whether that statement belongs corresponding slice or not. Note that such a design is not suitable for dynamic slicing, in which case, one would explore only one path to/from the slicing criterion.

5 NS-SLICER: NEURAL NETWORK-BASED STATIC PROGRAM SLICING

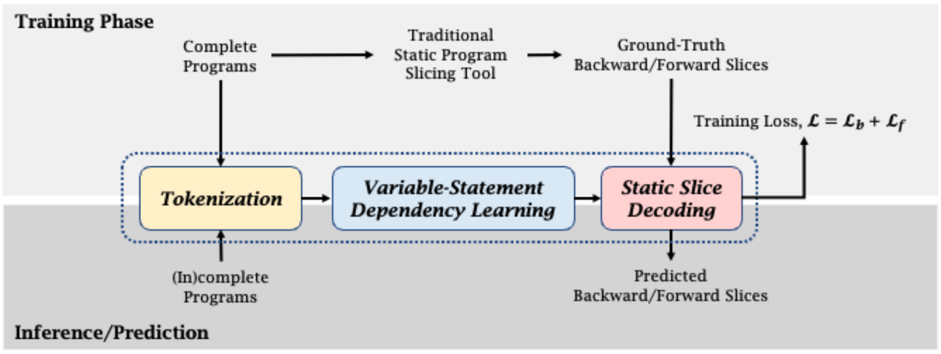

Training Phase
Complete Programs
Traditional Static Program Slicing Tool
Ground-Truth Backward/Forward Slices
Training Loss, $\mathcal{L} = \mathcal{L}_b + \mathcal{L}_f$
Tokenization
Variable-Statement Dependency Learning
Static Slice Decoding
(In)complete Programs
Predicted Backward/Forward Slices
Inference/Prediction

Fig. 4. Overview of NS-Slicer, a learning-based approach to static program slicing

5.1 An Overview of NS-Slicer

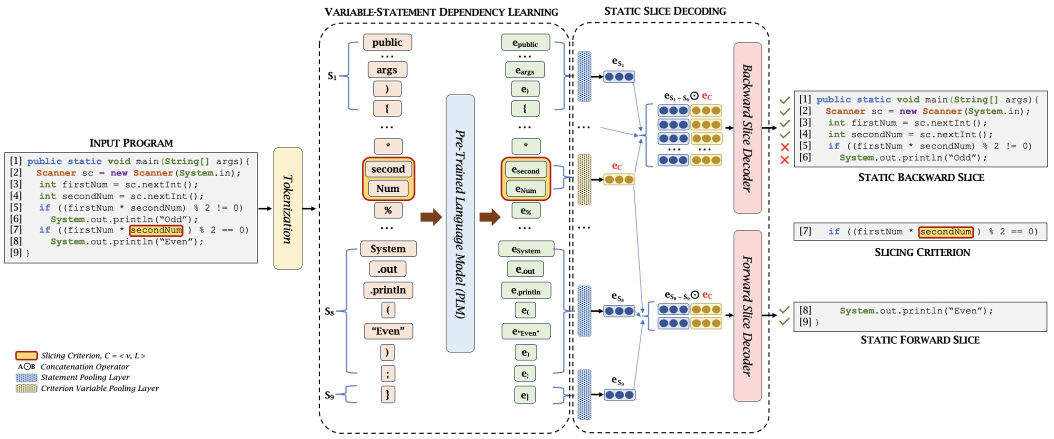
In Fig. 4, we present a general overview of NS-Slicer, a learning-based approach to static program slicing, which primarily has two phases: *training* and *inference*. The training phase yields a neural model that, given a complete/partial program and a slicing criterion, predicts the corresponding backward and forward slices. To this end, we train the model on ⟨*Program*, *Slicing Criterion*, *Backward/Forward Slice*⟩ tuples, wherein, the ground-truth program slices are extracted using a traditional static program slicing tool. Thus, the training phase necessitates that the collected programs are complete. During inference/prediction, possibly incomplete code snippets can be input to NS-Slicer.

Given a program, in both training and inference phases, we: *first*, split the source code into a sequence of sub-tokens; *second*, identify the variable-statement dependencies that encapsulate the knowledge of the variables across statements that either influence, or are influenced by the slicing criterion; *third*, construct the static program slices by discarding the irrelevant statements in the program – identifying the sets of statements that affect, or are affected by the slicing criterion.

5.2 Model Architecture

Given a complete or partial program and a variable at a specific location as the slicing criterion, NS-Slicer predicts the set of all statements that affect the variable in the requested criterion as the backward slice, and the set of all statements that can be affected by the variable as the forward slice. Fig. 5 illustrates the architecture of NS-Slicer, which has the following essential components:

5.2.1 Variable-Statement Dependency Learning. Several source code-specific PLMs [Feng et al. 2020; Guo et al. 2021] have emerged in recent time, which treat source code as a sequence of tokens. There are multiple benefits to incorporating PLMs into our framework. Firstly, prior research [Guo et al. 2021; Hernández López et al. 2023] demonstrates the ability of these advanced models to



```
[1] public static void main(String[] args){
[2]   Scanner sc = new Scanner(System.in);
[3]    int firstNum = sc.nextInt();
[4]    int secondNum = sc.nextInt();
[5]    if ((firstNum * secondNum) % 2 != 0)
[6]      System.out.println("Odd");
[7]    if ((firstNum * secondNum ) % 2 == 0)
[8]      System.out.println("Even");
[9] }
```

Static Backward Slice:

```
✓ [1] public static void main(String[] args){
✓ [2]   Scanner sc = new Scanner(System.in);
✓ [3]    int firstNum = sc.nextInt();
✓ [4]    int secondNum = sc.nextInt();
✗ [5]    if ((firstNum * secondNum) % 2 != 0)
✗ [6]      System.out.println("Odd");
```

Slicing Criterion:

```
[7]    if ((firstNum * secondNum ) % 2 == 0)
```

Static Forward Slice:

```
✓ [8]      System.out.println("Even");
✓ [9] }
```

Fig. 5. Illustration of framework architecture of NS-Slicer

capture both the syntax and semantics of programming languages. Through the training process, we expect that this component will leverage such knowledge of fine-grained dependencies in source code to learn to apply the flow analyses and identify the variable-statement dependencies. Moreover, the sequential nature of PLMs ensures NS-Slicer can operate on both complete and partial code, thus overcoming the limitations of traditional program slicing approaches, which work only on complete code.

Consider a program $P = \langle s_1, s_2, \cdots, s_N \rangle$, where s_i represents the statement comprising the sequence of code tokens $\langle c^i_1, c^i_2, \cdots, c^i_{M_i} \rangle$ on the i-th line (i.e., a total of M_i tokens). Accordingly, we input the sequence of code tokens $\langle c^1_1, c^1_2, \cdots, c^1_{M_1}, \cdots, c^N_1, c^N_2, \cdots, c^N_{M_N} \rangle$ to the PLM. For each of the tokens c^i_m in P, the PLM yields a rich and contextualized token representation $e_{c^i_m} \in \mathbb{R}^d$.

5.2.2 Pooling Layers. A pooling operation is useful to aggregate a sequence of token representations. In NS-Slicer, we have two pooling components: *variable pooling layer*, and *statement pooling layer*. The former computes the representation for the variable at the slicing criterion (i.e., *slicing variable embedding*) from the token representations of the comprising individual tokens within that variable. The latter computes the representation for a program statement (i.e., *statement embedding*) from the token representations of the comprising individual tokens within the statement. We use *mean-pooling* as the default aggregation strategy within both pooling layers.

Consider the variable $v = \langle c^x_{a+1}, \cdots, c^x_{a+b} \rangle$ on the x-th line that corresponds to the slicing criterion, and the statement s_i, that needs to be checked whether it belongs to the static program slice or not. Their pooled representations would be $e_v = \frac{1}{b} \sum_{j=a+1}^{a+b} e_{c^x_j}$ and $e_{s_i} = \frac{1}{M_i} \sum_{j=1}^{M_i} e_{c^i_j}$, respectively.

5.2.3 Static Slice Decoding. The *static slice decoding* phase has two components: *backward slice decoder*, and *forward slice decoder*. Both components construct the static backward and forward slices by deleting the parts of the program that do not affect, or are not affected by the variable at the slicing criterion, respectively. This is akin to traversing the PDG in the backward and forward directions to compute the corresponding program slices, except that, the knowledge of the *variable-statement dependencies* required for deriving the program slices is encoded in the statement representations e_{s_i}. We capture the nuanced difference between the static slice decoding components by leveraging two distinct 2-layered multi-layered perceptrons (i.e., MLP$_b$ for backward slicing and MLP$_f$ for forward slicing). To inspect whether a statement s_i in a given program P is involved in the computation of the variable v on the x-th line or not, we score the MLPs as follows:

$$\text{score}_{slice}(s_i, v_x) = \text{MLP}_{slice}(e_{s_i} \circ e_{v_x}) \tag{1}$$

where "$\circ$" corresponds to the concatenation operation, and $slice \in \{b, f\}$. If a statement s_i attains a $\text{score}_{slice}(s_i, v_x) > 0.5$, it indicates that s_i belongs to the static backward/forward slice of the variable v on line x in the given program P. The combination of all such statements represents the predicted static program slice, wherein, the backward slice $\hat{B} = \{s_i \mid s_i \in P \wedge \text{score}_b(s_i, v_x) > 0.5, 1 < i < x\}$ and forward slice $\hat{F} = \{s_i \mid s_i \in P \wedge \text{score}_f(s_i, v_x) > 0.5, x < i < N\}$. Note that MLP_b is applied to all statements preceding the slicing criterion $(1 < i < x)$ and MLP_f is applied to the ones following it $(x <$ i $< N)$. Thus, both sets of statements adjudged to belong to these slices are disjoint.

5.3 Training Process

We train NS-Slicer on complete programs to facilitate the extraction of ground-truth static program slices. We can leverage any traditional static program slicing technique [Tip 1995] to build such ground truth slices from complete code. Next, we will explain the mathematical formulation for the task of neural program slicing.

For the program $P = \langle s_1, s_2, \cdots, s_N \rangle$ and variable v on the x-th line as the slicing criterion, let the ground-truth backward and forward slices be $B = \langle y_1, y_2, \cdots, y_{x-1} \rangle$ and $F = \langle y_{x+1}, y_{x+2}, \cdots, y_N \rangle$, respectively. It is worth noting that the backward slicing classifier is applied to all statements preceding the x-th line, and the forward slicing classifier is applied to all the statements following it – the label y_i meaning that it belongs to either of them or not. Due to its formulation as a binary classification problem, the ground-truth backward slices B_j for all complete programs P_j in the training data D can be utilized to compute the training loss for backward slicing (i.e., $\mathcal{L}_b$) as:

$$\mathcal{L}_b(\theta) = \sum_{P_j \in D} \sum_{i=1}^{x_j - 1} \left\{ y_i \log(p^b(s_i, y_i) + (1 - y_i) \log(1 - p^b(s_i, y_i)) \right\} \tag{2}$$

Similarly, the ground-truth forward slices F_j for all the complete programs P_j in D can be utilized to compute the training loss corresponding to forward slicing (i.e., $\mathcal{L}_f$) as:

$$\mathcal{L}_f(\theta) = \sum_{P_j \in D} \sum_{i=x_j+1}^{N} \left\{ y_i \log(p^f(s_i, y_i) + (1 - y_i) \log(1 - p^f(s_i, y_i)) \right\} \tag{3}$$

The final joint training loss for NS-Slicer can be computed as follows:

$$\mathcal{L}(\theta) = \min_\theta \left\{ \mathcal{L}_b(\theta) + \mathcal{L}_f(\theta) \right\} \tag{4}$$

6 EMPIRICAL EVALUATION

We conducted several experiments to evaluate NS-Slicer, aiming to answer the following questions:

(I) Intrinsic Evaluation

RQ$_1$. Effectiveness on Complete Java Code: *How accurate is NS-Slicer in constructing program slices for a given criterion, i.e., a variable on a specific statement in complete Java programs?*

RQ$_2$. Effectiveness on Partial Java Code: *How accurate is NS-Slicer in constructing program slices for a given criterion, i.e., a variable on a specific statement in incomplete Java code?*

(II) Ablation Study

RQ$_3$. *How does the removal/replacement of its components affect NS-Slicer's model performance?*

(III) Qualitative Evaluation

RQ$_4$. Variable Aliasing: *Can NS-Slicer correctly predict the program slice for a Java program in which Java aliases have been introduced? How does it affect the overall model performance?*

(IV) Extrinsic Evaluation

RQ$_5$. Vulnerability Detection in Java Code: *How useful are the program slices built by NS-Slicer for the downstream task of detecting vulnerabilities in incomplete code?*

6.1 Static Slicing of Complete Programs (RQ$_1$)

6.1.1 Data Collection. To enable the effectiveness evaluation on complete Java programs (RQ$_1$), we considered IBM's Project CodeNet dataset [Puri et al. 2021], which includes 4,053 programming challenges for several languages. In particular, we focus on Java language as the main resource, which covers 250 problems and a total of 75,000 solutions. We parsed all such Java programs using Eclipse JDT to identify the locations of different variables in the programs. Next, we leveraged JavaSlicer [Galindo et al. 2022] to collect the static program slices for all such instances.

Since we model the program slice decoding task as binary classification, it is imperative to ensure that the labels for positive and negative samples are balanced. Thus, we only retain those instances in which the ratio (r) of the statements belonging to the backward/forward slice, to the ones not belonging to the corresponding slice lies between 0.3 and 0.7, i.e., $0.3 < r < 0.7$. r is the range for the ratio between the number of positive and negative samples (statements) that are input to the slicing classifiers. Note that this ratio does not correspond to the size of the slice over the program's size. By not enforcing a form of balance of the positive and negative samples in this manner, the classifiers would not capture well the notion of belonging, or not belonging to the slice. Accordingly, we retain ~43,000 data instances corresponding to the Java programs, each containing between 5–69 statements per program. Finally, we split them at the problem-level in a 80%/10%/10% ratio, i.e., dedicating the Java programs and their slices across 200 problems for training, 25 problems for validation, and 25 problems for testing, respectively. Such a problem-level splitting strategy avoids data corruption across the dataset splits, thus better representing a realistic scenario.

6.1.2 Methodology. Pre-trained language models (PLMs) on source code benefit from the pre-training tasks by learning to encode the whole structure in programming languages [Hernández López et al. 2023], and the semantic-level structure in code [Guo et al. 2021]. In particular, the data flow-specific pre-training objective in GraphCodeBERT [Guo et al. 2021] encodes in it the relation of where the value in a variable comes from, making it suitable for our task of static program slicing. Thus, we leveraged GraphCodeBERT as the PLM in NS-Slicer. During the training phase, we fine-tuned it alongside training the MLP heads corresponding to backward and forward slicing, respectively.

By virtue of design, the intrinsic evaluation (RQ1–RQ2) assesses how closely NS-Slicer *mimics the traditional program slicing tool, i.e., JavaSlicer* in terms of resulting slices. We also picked multiple baselines to compare against our model. First, we chose the pre-trained version of GraphCodeBERT off-the-shelf. In this case, during the training phase of NS-Slicer, the parameters within the PLM were fixed, and the token/statement representations from the PLM were used as-is, to train the backward and forward slicing MLP heads. Next, we chose the state-of-the-art CodeBERT model [Feng et al. 2020], considering both the pre-trained and fine-tuned versions as above.

The backbone of the PLMs in NS-Slicer is the standard RoBERTa-base architecture [Liu et al. 2019] which has 12 Transformer-encoder layers, each possessing 12 attention heads. We used the byte-pair encoding (BPE) [Liu et al. 2019] scheme in the pre-trained `RobertaTokenizer` for splitting the given source code into sub-tokens, with its corresponding initialization as per the NS-Slicer variant. The dimension size for the token representations thus produced by these PLMs is 768. All our experiments were conducted on an NVIDIA RTX A6000 GPU. With a batch size of 64, we initialized all variants with learning rates of 1e-4 and 5e-4 during training, and reported the

Table 1. Effectiveness evaluation on complete Java programs (RQ$_1$).

Approach		Slicing Criterion	Evaluation Metrics (in %)					
			A-EM	*A-D*	*A-S*	*P*	*R*	*F1*
Pre-Trained	CodeBERT	Backward	43.21	92.24	84.40	88.52	88.30	88.41
		Forward	40.84	94.80	85.76	83.62	87.20	85.37
		Overall	**42.03**	**92.46**	85.06	86.51	87.86	**87.18**
	GraphCodeBERT	Backward	47.53	92.54	86.91	91.75	88.54	90.12
		Forward	49.75	95.76	88.07	86.03	89.49	87.72
		Overall	**48.64**	**93.15**	87.47	89.36	88.92	**89.14**
Fine-Tuned	CodeBERT	Backward	81.72	97.72	95.65	97.01	96.52	96.76
		Forward	83.47	97.88	95.59	95.31	95.45	95.38
		Overall	**82.59**	**97.56**	95.62	96.33	96.09	**96.21**
	GraphCodeBERT	Backward	85.77	98.21	96.51	97.60	97.21	97.41
		Forward	84.62	98.49	95.99	95.20	96.45	95.82
		Overall	**85.20**	**98.09**	96.26	96.63	96.91	**96.77**

best-performing models. Overall, NS-Slicer (with both CodeBERT and GraphCodeBERT) has about 128M parameters, and took ~32 minutes per epoch to train on our machine.

6.1.3 Evaluation Metrics. We model the program slice decoding step in NS-Slicer as a binary classification problem at each statement. Thus, we adopt the standard evaluation metrics: *Accuracy-S* $= \frac{\text{TP+TN}}{\text{TP+FP+FN+TN}}$, *Precision* $= \frac{\text{TP}}{\text{TP+FP}}$, *Recall* $= \frac{\text{TP}}{\text{TP+FN}}$, and *F1-Score* $= \frac{2*Precision*Recall}{Precision+Recall}$. Here, TP = True Positives, FP = False Positives, FN = False Negatives, and TN = True Negatives. Note that we enable the statement-level assessment by counting all the outcomes in these metrics globally.

Next, to capture the model performance at the slice-level, we report the stricter Exact-Match Accuracy (i.e., *Accuracy-EM*) that determines the number of times NS-Slicer predicts the backward and forward slices exactly the same as the ground-truth slices from JavaSlicer.

Finally, we report Dependence Accuracy (i.e., *Accuracy-D*) to assess how accurately the inter-statement dependencies are predicted, causing a particular statement to be included in the slice. Accordingly, we define *Accuracy-D* for a particular program as the ratio of the correctly predicted dependencies to the actual dependencies across all slicing criteria for that program, finally reporting the mean across all programs in the dataset. We use the same set of metrics for RQ1–RQ4.

6.1.4 Experimental Results. In Table 1, we report the performance of NS-Slicer on complete Java programs, comparing different PLMs for the static program slicing task. We can see that using GraphCodeBERT (rows 10–12) produces the most competitive results, predicting the backward and forward slices with an F1-score of 97.41% and 95.82%, respectively. Overall, our tool best predicts the complete static slice with an F1-score of 96.77%. Furthermore, it matches the ground-truth program slices constructed by JavaSlicer exactly, in 85.20% of the cases. Note that these ground-truth program slices are executable, i.e., are syntactically valid and can be executed independently of the main program. Thus, 85.20% of the program slices predicted by NS-Slicer are also executable, demonstrating its usefulness in debugging real-world Java programs.

CodeBERT is pre-trained on the masked-language modelling (MLM) and replaced-token detection (RTD) learning objectives, compared to GraphCodeBERT, which leverages code structure and data-flow for pre-training. As a result, using GraphCodeBERT in place of CodeBERT as the PLM in NS-SLICER results in a relative improvement in the overall F1-score by 0.58%, and exact-match accuracy

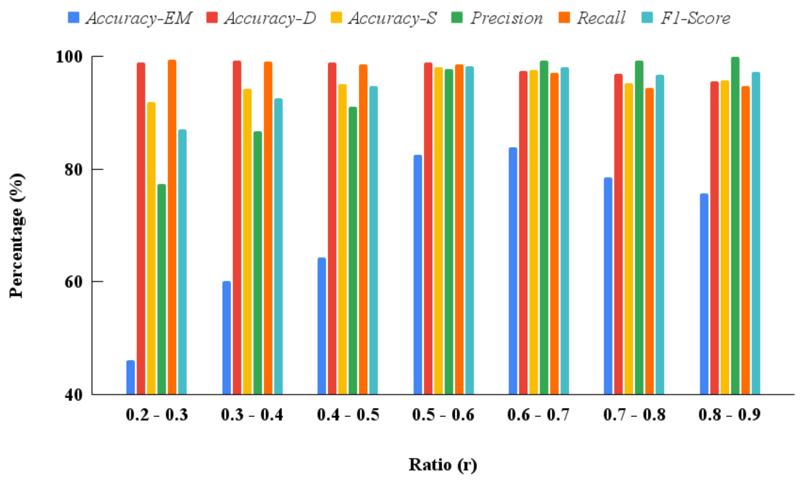


Fig. 6. Sensitivity to the static program slice sizes as a fraction of the entire program.

by 3.16%. Besides, the improvement in predicting such matched slices can possibly be attributed to the data-flow knowledge in GraphCodeBERT, which ensures both statements containing the variable belong to the slice. *We conducted t-test between CodeBERT and GraphCodeBERT PLMs in NS-Slicer, and the results show the improvements are significant with* $p < 0.01$.

We also report the performance of NS-Slicer by leveraging CodeBERT and GraphCodeBERT PLMs off-the-shelf (rows 1–3 and 4–6 in Table 1). Here, only the backward and forward slicing MLP-heads are trained. The results demonstrate a decrease in the overall F1-score by 11% and 8.56%, respectively. Moreover, there is a notable decline in the exact-match accuracy, with a drop of 102.71% for CodeBERT and 75.16% for GraphCodeBERT. Such a drop in performance *reinforces the complexity of the program slicing task* and underscores the rationale behind NS-Slicer's design.

As seen in Table 1, NS-Slicer also achieves very high Dependence Accuracies (*Accuracy-D*) across all the models from 92.46%–98.09%. This validates our hypothesis on the ability of the PLMs in learning the variable-statement dependencies.

NS-Slicer predicts the program slices for complete Java programs with an overall F1-score of 96.77%, and exactly matches the ground-truth program slices in 85.20% of the cases.

6.1.5 Sensitivity to the Size of Static Program Slices. In Fig. 6, we report the stratified performance of NS-Slicer with fine-tuned GraphCodeBERT based on the ratio (r) of the sizes of static program slices to the corresponding lengths of the programs. Here, note that r in the given ranges (i.e., the range of 0.2–0.3 to the range of 0.8–0.9) corresponds to 1.1%, 10.7%, 17.8%, 23.5%, 21.8%, 19.1%, and 5.8% of the test set, respectively.

A lower value of r indicates fewer variable-statement dependencies in the program. Notably, when $0.2 < r < 0.5$, we can see that the dependence accuracy and recall are among the highest. This showcases the effectiveness of NS-Slicer in identifying the presence of such dependencies. However, based on the lower precision in these cases, we can conclude that it does not identify the statements that do not belong to the static program slice as well. As a result, the exact-match accuracy and F1 scores are also lower. We can attribute this skewedness to the fewer number of

Table 2. Effectiveness evaluation on partial Java programs, where $P\%$ is omitted at both start and end (RQ$_2$)

P	**Slicing Criterion**	**Evaluation Metrics (in %)**					
		A-EM	*A-D*	*A-S*	*P*	*R*	*F1*
5%	Backward	85.98	98.26	96.67	97.57	97.40	97.49
	Forward	80.99	97.17	95.59	95.25	95.56	95.40
	Overall	**83.48**	**97.76**	96.14	96.60	96.64	**96.62**
10%	Backward	83.89	98.10	96.17	97.08	96.91	96.99
	Forward	72.91	93.79	94.12	95.45	92.68	94.05
	Overall	**78.40**	**96.13**	95.14	96.37	95.03	**95.70**
15%	Backward	84.54	97.31	95.91	97.32	96.23	96.77
	Forward	62.02	89.89	91.83	96.00	88.57	92.14
	Overall	**73.28**	**93.92**	93.85	96.73	92.68	**94.66**

examples with a low value of r during training. Moreover, the nature of static program slicing diminishes the consequences of including such non-dependent statements in the slice.

6.2 Static Slicing of Partial Programs (RQ$_2$)

Traditional program analysis approaches fail to extract static program slices for partial programs (for reasons shown in Section 3). In contrast, NS-Slicer is not limited by the program's completeness. We set up this experiment to exhibit its ability in predicting static slices for partial Java code.

6.2.1 Methodology. Extracting the ground-truth program slices for partial programs is not possible. As a result, for the test Java programs in Section 6.1, we strip $P\%$ of the statements at the beginning and at the end of the code example to imitate a partial Java program. To evaluate the performance of NS-Slicer on partial code thus obtained, we extract the ground-truth partial program slices corresponding to the retained code, from the ground-truth program slices of complete code. In this process, we omit any instances which end up with empty backward or forward slices. The difficulty in dealing with partial programs obtained in such a manner is exacerbated by the possible deletion of various variable declarations – thus eliminating the *def-use* relationships. Moreover, the higher the value of P, more number of such relationships shall be defied.

6.2.2 Experimental Results. In Table 2, we report NS-Slicer's results on partial Java code obtained by omitting 5%, 10%, and 15% of the programs, both at the beginning and the end. We record an overall F1-score of 94.66%–96.62%, and an exact-match accuracy of 73.28%–83.48%. Overall, with this stripping scheme, an average of 2, 3, and 5 statements per program are omitted, respectively. Notably, in 14.89%, 51.97%, and 59.46% of the cases, the excluded statements contain variable declarations that are referenced in the remaining partial program, thus disrupting the *def-use* chain. Thus, we can explain the decrease in performance by 2.07% in overall F1-score and 13.92% in exact-match accuracy, as resulting from the deletion of 5%$\longrightarrow$ 15% of the program.

Comparing to the Dependence Accuracies (*Accuracy-D*) in Table 1, the corresponding accuracies for partial code in Table 2 are lower. However, they are all from 93.92% to 97.76%, validating our hypothesis of the learning capability of PLMs on variable-statement dependencies.

NS-Slicer predicts the program slices for partial Java code with an F1-score of 94.66%–96.62%, and exactly matches the ground-truth program slices in 73.28%–83.48% of the cases.

Table 3. Ablation Study: Here, B_1 denotes NS-Slicer *w/o source code pre-training*; B_2 denotes NS-Slicer *w/o data-flow pre-training*; B_3 denotes NS-Slicer *w/o mean-pooling* (RQ$_3$).

Baseline	**Evaluation Metrics (in %)**					
	A-EM	*A-D*	*A-S*	*P*	*R*	*F1*
B_1	82.19	97.84	95.50	96.60	95.59	96.09
B_2	83.30	97.71	95.79	96.66	96.03	96.35
B_3	80.15	96.57	95.41	96.65	95.36	96.00
NS-Slicer	85.20	98.09	96.26	96.63	96.91	96.71

6.3 Ablation Study (RQ$_3$)

In this experiment, we aim to quantify the contributions of different model components in NS-Slicer to its overall performance.

6.3.1 Ablation Baselines. The different ablation baselines include:

- B_1, *w/o source code pre-training*: PLMs such as CodeBERT, GraphCodeBERT, *etc.*, are trained on source code from multiple programming languages, and have exhibited benefits to tasks such as clone detection, code summarization, *etc.* The underlying RoBERTa-base [Liu et al. 2019] model architecture, however, was originally trained on multiple English-language corpora. We created this baseline to observe the effect that such source code pre-training has on the static program slicing task. Thus, we initialize the PLM in NS-Slicer with the *pre-trained RoBERTa-base* model.
- B_2, *w/o data-flow pre-training*: The *masked language modeling* (MLM) pre-training objective has been shown to help source code PLMs understand the structure in programming languages [Hernández López et al. 2023]. In GraphCodeBERT, Guo *et al.* [Guo et al. 2021] included additional pre-training tasks such as *edge prediction* and *node alignment* to help the model learn where the value in a variable comes from, i.e., data-flow. We created this ablation baseline to understand the affect that these pre-training objectives have on the data flow-specific task of program slicing. Thus, we initialize the PLM in NS-Slicer with *CodeBERT model pre-trained only on MLM.*
- B_3, *w/o mean-pooling*: In NS-Slicer, by default, we use the *mean-pooling* strategy within the *Variable Pooling Layer* and *Statement Pooling Layer* to aggregate the contextualized sub-token representations produced by the PLMs for the corresponding sub-tokens. We created this baseline to assess the effect of such an aggregation, and replace it with *max-pooling* instead.

6.3.2 Experimental Results. In Table 3, we report the performance of the different ablation baselines B_1–B_3. To establish B_1, we replace the source code-based PLMs in NS-Slicer with the RoBERTa-base model pre-trained on the English language. Interestingly, we can see that the lack of source code-based pre-training only results in a drop in performance from using the GraphCodeBERT PLM in NS-Slicer by 0.62% in overall F1-score and 3.01% in exact-match accuracy. Nevertheless, the disparity between B_1, which lacks source code understanding, and the use of source code-based PLMs in NS-Slicer becomes evident when tested on partial code (as in Section 6.2). In this case, B_1 observes an exact-match accuracy of 68.54%–80.33%, highlighting the advantages of utilizing source code-based PLMs in our solution.

With B_2, we aim to isolate the benefits of leveraging the data flow-specific pre-training objectives in GraphCodeBERT for program slicing. We see that using just the MLM learning objective results in a relative drop in performance by 0.37% in overall F1-score and 1.9% in exact-match accuracy. Note that B_2 is different from the pre-trained CodeBERT in Table 1 (rows 7–9), as it is not trained on the RTD learning objective. In this case, it improves upon CodeBERT pre-trained on both MLM and RTD learning objectives by 0.15% in overall F1-score and 1.22% in exact-match accuracy.

Devlin *et al.* [Devlin et al. 2019] assert that each downstream task benefits from different sub-token aggregation techniques. In B_3, we test *max-pooling* instead of *mean-pooling*, and observe that the overall F1-score drops by 0.71% and the exact-match accuracy drops by 5.1%. In fact, it is the worst-performing ablation baseline among B_1–B_3. Therefore, we can quantifiably measure the advantage of employing *mean-pooling* for extracting the contextualized variable and statement representations in NS-Slicer to predict the corresponding program slices.

From Table 3, we learned that mean-pooling (B_3), to obtain variable/statement representations is more critical in our design. This is reflected by the 5.1% drop in Exact-Match score upon replacing it by max-pooling, another popular sub-token aggregation strategy. Comparing B_1 and B_2, we saw that source code pre-training is more important than data flow pre-training. Furthermore, the importance of source-code-pre-training is exacerbated in the case of partial code. As seen, NS-Slicer's performance dropped significantly from 73.28% (Table 2) to 68.54%.

All model components in NS-Slicer directly contribute to its ability in predicting highly accurate static program slices.

6.4 Probing Pre-Trained Language Models (PLMs) for Variable Aliasing (RQ$_4$)

Variable aliasing is an intrinsic code property in which multiple references when used to refer to the same object, still point to the same location in memory. In practice, aliasing is not straightforward to debug, as the changes caused on one variable also reflects on the other. We set up this experiment to assess how well the PLMs understand variable aliasing, and the effect such aliases have on the performance of NS-Slicer in static program slicing.

6.4.1 Methodology. We enable this experiment by introducing synthetic variable aliases in the test Java programs in Section 6.1, by inserting a variable assignment in the statement following the one containing the variable of the requested criterion. Next, in all the subsequent statements that contain the variable, we replace the original variable with the alias. For example, let us consider a Java program in which we need to predict the program slice for the variable `x` of type `int`, on statement 5. Let the forward slice for this contain the statements 8, 9, 13, and 15, among which statements 9 and 13 contain `x`. In this case, we introduce a variable assignment `int aliasingVar=x` on statement 6, and replace `x` on statements 9 and 13 by `aliasingVar`. As a result, the new forward slice contains the statements 6, 9, 10, 14, and 16, while the backward slice remains unaffected. Thus, for the test programs obtained in such a manner, we compare the forward slicing performance of CodeBERT and GraphCodeBERT in NS-Slicer, with and without aliasing.

6.4.2 Experimental Results. In Table 4, we report the forward slicing performance of both CodeBERT and GraphCodeBERT PLMs, on test programs with and without synthetic variable aliases. We observed a drop in the forward slicing F1-score, with reductions of only 13.32% and 12.31%, respectively. This shows that NS-Slicer is able to track the dependencies across the variable aliases (in Section 6.4.3, we demonstrate such dependency tracking via a case study). In contrast, the forward slicing exact-match accuracy shows substantial declines, with a drop of 139.72% for CodeBERT and 118.43% for GraphCodeBERT. The lower exact-match accuracy, despite having a high F1-score, can be attributed to the strictness of the former metric. This suggests that NS-Slicer may have mis-predicted only a few statements per program.

We performed an in-depth analysis of the test Java programs to gain more insights about this behavior. On average, 42.86% of the statements in the forward slice for each program contains

Table 4. Probing Pre-Trained Language Models (PLMs) for Variable Aliasing: Comparison of forward slicing performance on complete Java programs, with (M_{x}) and without ($M^{*}{}_{\text{x}}$) synthetic variable aliases (RQ$_4$).

Approach	**Evaluation Metrics (in %)**					
	A-EM	*A-D*	*A-S*	*P*	*R*	*F1*
M_{CodeBERT}	34.82	74.33	81.31	90.12	78.95	84.17
$M^{*}{}_{\text{CodeBERT}}$	83.47	97.88	95.59	95.31	95.45	95.38
$M_{\text{GraphCodeBERT}}$	38.74	74.23	82.49	90.27	80.89	85.32
$M^{*}{}_{\text{GraphCodeBERT}}$	84.62	98.49	95.99	95.20	96.45	95.82

references to the synthetic variable `aliasingVar`. Next, to investigate the effect of the number of aliasing variable references on NS-Slicer's performance, we stratified the test set based on the number of such references and compared the forward slices. However, we did not observe any direct correlation between the number of references and our tool's predictive capabilities.

While the predicted forward slices still hold value, the PLMs may encounter challenges in generating *exact-match slices* in such complex yet uncommon aliasing scenarios. This could be attributed to the lack of understanding of source code at the memory level, as *the current pre-training tasks mainly focus only on the lexical aspects of source code.* Thus, advancements in source code-specific pre-training could further improve the performance of PLMs for program slicing.

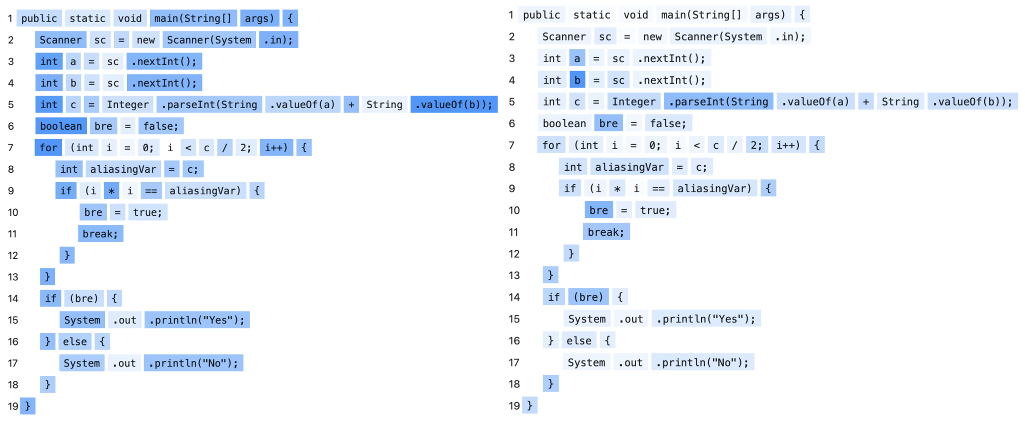
The PLMs in NS-Slicer predict the forward slices for complete Java programs with variable aliases with an F1-score of 84.17%–85.32%, and exact-match accuracy of 34.82%–38.74%.

6.4.3 *Case Study 1.* The self-attention [Vaswani et al. 2017] mechanism in PLMs facilitates contextualization by assigning attention scores to each token based on its relationship with all the other tokens within its context. Thus, for an input containing N tokens, we obtain an $N \times N$ attention map, where each attention score numerically signifies the importance of one token on the other. In this section, we leverage such attention maps to investigate the GraphCodeBERT PLM's understanding of *variable-statement dependencies* within NS-Slicer.

In Fig. 7, we present a test Java program, for which the static program slices need to be derived with respect to variable `c` on line 7. The ground-truth backward and forward slices for this slicing criterion include the lines $\{1, 2, 3, 4, 5\}$ and $\{8, 9, 11, 12, 13, 19\}$, respectively. To predict the static program slices, we leverage two versions of NS-Slicer from Section 6.1, utilizing the *pre-trained* GraphCodeBERT (row 2 in Table 1), and *fine-tuned* GraphCodeBERT (row 4 in Table 1).

Fig. 7a and Fig. 7b correspond to the *pre-trained* and *fine-tuned* GraphCodeBERT PLMs within NS-Slicer, respectively. In both, we illustrate the words (comprising tokens separated by a whitespace or a period) in the test Java program in the form of a heatmap – highlighting the attention scores of all words with respect to the variable `c` on line 7. To compute the attention score for a word in the heatmaps, we follow prior work [Clark et al. 2019] and take the mean of all such scores over its sub-tokens. In general, a darker colour in the heatmaps indicate a stronger relationship with variable `c`, in the form of *variable-statement dependencies*. Moreover, given that NS-Slicer is geared to predict whether a statement belongs to the program slice with respect to a variable in a requested criterion or not, the attention scores here suggest such decision making.

When leveraging the *pre-trained* GraphCodeBERT (Fig. 7a), we can see that the model does not pay attention to the data and control dependent variables. As a result, the program slice computed by plugging the pre-trained GraphCodeBERT model in NS-Slicer is not capable of predicting



(a)

```
1  public static void main(String[] args) {
2    Scanner sc = new Scanner(System .in);
3    int a = sc .nextInt();
4    int b = sc .nextInt();
5    int c = Integer .parseInt(String .valueOf(a) + String .valueOf(b));
6    boolean bre = false;
7    for (int i = 0; i < c / 2; i++) {
8      int aliasingVar = c;
9      if (i * i == aliasingVar) {
10       bre = true;
11       break;
12     }
13   }
14   if (bre) {
15     System .out .println("Yes");
16   } else {
17     System .out .println("No");
18   }
19 }
```

(b)

```
1  public static void main(String[] args) {
2    Scanner sc = new Scanner(System .in);
3    int a = sc .nextInt();
4    int b = sc .nextInt();
5    int c = Integer .parseInt(String .valueOf(a) + String .valueOf(b));
6    boolean bre = false;
7    for (int i = 0; i < c / 2; i++) {
8      int aliasingVar = c;
9      if (i * i == aliasingVar) {
10       bre = true;
11       break;
12     }
13   }
14   if (bre) {
15     System .out .println("Yes");
16   } else {
17     System .out .println("No");
18   }
19 }
```

Fig. 7. Visualization of attention score heatmaps from *pre-trained* (left) and *fine-tuned* (right) GraphCodeBERT PLMs within NS-Slicer, for all words in a Java program, sliced with respect to variable c on line 7.

```
1   public static void main(String[] args) throws Exception {                              ✓
2     Scanner sc = new Scanner(System.in);                                                  ✓
3     int n = sc.nextInt();                                                                 ✓
4     String ans = "Yes";                                                                   ✗
5     int p_t = 0;                                                                          ✓
6     int p_x = 0;                                                                          ✗
7     int p_y = 0;                                                                          ✗
8     for(int i = 0; i < n; i++){                                                           ✓
9       int t = sc.nextInt();                                                               ✓
10      int x = sc.nextInt();                                                               ✓
11      int y = sc.nextInt();                                                               ✓
12      int diff = Math.abs(x - p_x) + Math.abs(y - p_y);                                   ✓
13      int aliasingVar = t;                                                                ✓
14      if(diff > aliasingVar - p_t || Math.abs(aliasingVar - p_t - diff) % 2 == 1){        ✓
15        ans = "No";                                                                       ✗
16        break;                                                                            ✓
17      }                                                                                   ✓
18      p_t = aliasingVar;                                                                  ✓
19      p_x = x;                                                                            ✗
20      p_y = y;                                                                            ✗
21    }                                                                                     ✓
22    System.out.println(ans);                                                              ✗
23  }                                                                                       ✓
```

Fig. 8. Java program sliced with respect to variable t on line 9: ✓ denotes statements correctly identified by NS-Slicer as part of the slice, ✗ indicates statements erroneously classified as not belonging to the slice.

accurate backward and forward slices. Therefore, we can see that PLMs cannot directly be used for the program slicing task, reiterating the need to formulate the program slicing task as in NS-Slicer.

In contrast, utilzing *fine-tuned* GraphCodeBERT in NS-Slicer correctly predicts the backward and forward slice for this program. In Fig. 7b, we see that it pays more attention to the variables `a` and `b` on line 5, and subsequently, also to the corresponding variable declarations on lines 3 and 4, respectively. Thus, it predicts the lines $\{3, 4, 5\}$ to belong to the backward slice. Applying a similar reasoning, we can also explain the prediction of line $\{2\}$ belonging to the backward slice. On line 8, the synthetic variable alias `aliasingVar` is introduced. As a result, lines $\{8, 9\}$ are predicted to belong to the forward slice. While lines 10 and 11 are control-dependent on line 9, we can see that the variable `bre` does not depend on either `c` or the alias `aliasingVar` anywhere else. Thus, line $\{6\}$ and lines $\{14, 15, 16, 17\}$ are omitted from the backward and forward slices, respectively. Due to the control dependency described earlier, lines $\{11, 12\}$; and due to the nature of executable slices produced by JavaSlicer in the ground truth, lines $\{13, 19\}$, are predicted to belong to the forward slice. Therefore, as noted in Key Idea 2 (Section 4), we can see that **NS-Slicer exhibits an understanding of the *variable-statment dependencies***. Furthermore, we observe significant program slicing-specific code understanding within the attention maps of the PLM in NS-Slicer.

6.4.4 Case Study 2. Next, we present a test Java program in Fig. 8 for which the ground-truth static backward and forward slices with respect to variable `t` on line 9 are $\{1, 2, 3, 5, 6, 7, 8\}$ and $\{10, 11, 12, 13, 14, 16, 17, 18, 19, 20, 21, 23\}$, respectively. We can see that the variable `ans` depends on the conditions related to the variables `t`, `p_t`, `diff`, `x`, and `y`. However, it does not directly contribute to the computation of `t` on line 9. As a result, the *fine-tuned* GraphCodeBERT (row 4 in Table 1 omits lines $\{4\}$ and $\{15, 22\}$ from the static backward and forward slices.

Furthermore, the `if`-condition on line 14 depends on the variable `diff`, which, in turn depends on variables `p_x` and `p_y`. Thus, the lines $\{6, 7\}$ and $\{19, 20\}$ should belong to the static backward and forward slices, respectively. With the introduction of the alias `aliasingVar` on line 13, NS-Slicer seems to miss this indirect dependency, leading it to omit these lines in the corresponding predicted slices. However, upon testing this program without `aliasingVar`, we observed that NS-Slicer predicts the static program slice correctly, confirming the intricacies of understanding variable aliasing.

6.5 NS-Slicer for Vulnerability Detection (RQ$_5$)

In this experiment, we explore the applicability of the static program slices predicted by NS-Slicer for the downstream task of vulnerability detection (VD) in Java code. VulDeePecker [Li et al. 2018] is a popular automated VD approach that utilizes program slices for this purpose – representing programs as *code gadgets* that are composed of a number of semantically related program statements. Each code gadget focuses on a *key point* hinting at the existence of a vulnerability.

In Fig. 9, we present the general overview of VulDeePecker. During the training phase, VulDeePecker approach utilizes a BiLSTM model to learn to detect the vulnerability patterns from the code gadgets. In the detection phase, it breaks a given program down into multiple code gadgets, leveraging the trained BiLSTM model to determine whether each is vulnerable or not.

Next, we present the pipeline for extracting such code gadgets in VulDeePecker [Li et al. 2018]. Technically, a code gadget is a static program slice from an arguments of API method call. In Fig. 10 (*top*), it: *first*, extracts all the libary/API call arguments; *second*, builds the PDGs using Joern and apply data flow analyses to extract the corresponding gadgets (static slices); *third*, retrieves the *vulnerability labels* for the code gadgets. However, its dependence on traditional PA approaches for building static program slices *limits VulDeePecker to complete code.* Alternatively, NS-Slicer can be plugged into the second step of the process for extracting code gadgets, as in Fig. 10 (*bottom*). That is, we used the slices predicted by NS-Slicer in both training and prediction phases of VulDeePecker. Such an integration extends VulDeePecker's applicability to both complete and partial code.

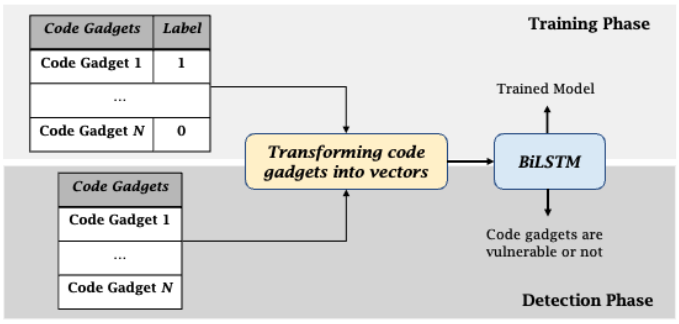


Fig. 9. Overview of VulDeePecker

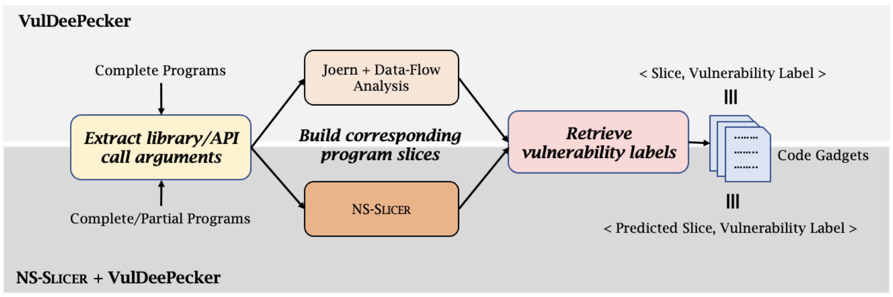

VulDeePecker
Complete Programs
Joern + Data-Flow Analysis
< Slice, Vulnerability Label >
Extract library/API call arguments
Build corresponding program slices
Retrieve vulnerability labels
Code Gadgets
Complete/Partial Programs
NS-Slicer
< Predicted Slice, Vulnerability Label >
NS-Slicer + VulDeePecker

Fig. 10. Contrasting *code gadget*-building pipelines in: VulDeePecker (*top*), NS-Slicer + VulDeePecker (*bottom*).

6.5.1 Data Collection and Procedure. Most benchmark vulnerability datasets cover C/C++ (e.g., BigVul [Fan et al. 2020]), and those for Java are limited. [Nikitopoulos et al. 2021] introduced CrossVul, a cross-language vulnerability dataset that includes vulnerable files written in more than 40 programming languages, covering 563 commits across Java files. We parse them using Eclipse JDT and extract a total of 14,139 methods with external libraries. Note that NS-Slicer can work on all of those methods regardless of whether they are complete (compilable) or not. We follow the methodology in BigVul [Fan et al. 2020] to label them as *vulnerable* or *non-vulnerable* – retrieving 574 and 13,565 for each, respectively. To ensure a balanced dataset, we only select files that are independent of the ones possessing the vulnerable methods, and randomly pick 574 non-vulnerable ones. Finally, we split them in a 80%/10%/10% train/validation/test ratio at the file-level.

Similar to [Li et al. 2018], we pick the library/API function call arguments within a method as the focus area. Accordingly, we collect all library/API call arguments for each method, determining their *vulnerability labels* based on whether the API call belongs to the changed code or not. As observed in Section 3, Joern fails to produce quality PDGs in the absence of external libaries, subsequently generating program slices with a low accuracy. Thus, we leverage NS-Slicer to extract the program slices corresponding to the API call arguments, and generate the code gadgets as explained.

Overall, we obtain 1,476 code gadgets for training, 184 for validation, and 187 for testing. Among these gadgets, 1,020 are vulnerable and 827 are non-vulnerable. To avoid underfitting of the BiLSTM in VulDeePecker due to the limited size of the Java vulnerability dataset, we utilize the version trained on BigVul [Fan et al. 2020] as the initialization point. Such a strategy leverages knowledge gained from predicting vulnerabilities in C/C++ code, leading to better generalization for Java code.

Table 5. Results of NS-Slicer + VulDeePecker for vulnerability detection, where M corresponds to the PLM.

NS-Slicer {M} + VulDeePecker	**Evaluation Metrics (in %)**			
	A	*P*	*R*	*F1*
Pre-trained GraphCodeBERT	59.46	64.71	62.86	63.77
Fine-tuned GraphCodeBERT	60.00	58.96	97.14	**73.38**

6.5.2 Evaluation Metrics. We adopt the same evaluation metrics as in prior vulnerability detection literature [Li et al. 2018; Wu et al. 2022], i.e, *Accuracy*, *Precision*, *Recall*, and *F1-Score*.

6.5.3 Experimental Results. In Table 5, we present the performance of VulDeePecker in predicting vulnerabilities via program slices, constructed by using the pre-trained GraphCodeBERT as the PLM in NS-Slicer (row 1), and the fine-tuned version of GraphCodeBERT from Section 6.1 (row 2). We can see that using the program slices from NS-Slicer helps predict vulnerabilities in such Java code with an *F1-score of 73.38%, while outperforming the baseline by 15.1%*. Such an improvement in performance further reiterates the role of learning *variable-statement dependencies* (See Section 4, Key Ideas 2–3), as opposed to only relying on data-flow knowledge of pre-trained GraphCodeBERT.

Furthermore, as Zhou *et al.* [Zhou and Sharma 2017] note, real-world vulnerabilities exist in a 9:1 non-vulnerable to vulnerable ratio. To test the performance of NS-Slicer + VulDeePecker in a real-world setting, we replicated this ratio across our test set and averaged the performance across 10 samples – achieving an F1-score of 98.67%. However, due to the significantly fewer number of vulnerable code gadgets in such samples, we note that the metrics in this setting are skewed.

The static program slices predicted by NS-Slicer help the automated vulnerability detection tool, VulDeePecker, detect vulnerabilities in partial Java code with an F1-score of 73.38%.

6.6 Limitations and Threats to Validity

6.6.1 Programming Languages. There do not exist many public, open-source, and non-proprietary static program slicing tools for different programming languages. For modern Java too, to the best of our knowledge, JavaSlicer [Galindo et al. 2022] is the only one. As a result, NS-Slicer primarily supports Java at this time. This framework can easily be adapted to other programming languages by collecting ⟨*Program*, *Slicing Criterion*, *Backward/Forward Slice*⟩ tuples as in Section 6.1. Moreover, the underlying PLM in NS-Slicer was originally trained on six programming languages, and can easily be adapted for enabling static program slicing for these languages by constructing those ⟨*Program*, *Slicing Criterion*, *Backward/Forward Slice*⟩ tuples.

6.6.2 External APIs. JavaSlicer currently fails to build static program slices for programs with external dependencies. We reported to the developers about the same, which will potentially be fixed later. As a result, in the current version, our dataset only covers programs with no external libraries. Hence, extending our tool to programs with external libraries (which will require manual program analysis for extracting the ground truth program slices) is left for future work.

6.6.3 Model Size. In this work, we only compare with CodeBERT and GraphCodeBERT PLMs in NS-Slicer, which internally make use of the RoBERTa-base model architecture – thus limiting the model input size to 512 tokens. However, our dataset still covers instances with an average of 19 statements per program. Besides, a PLM with a larger input size, for example, LongCoder [Guo et al. 2023] (maximum of 4,096 tokens) can easily be plugged into our framework.

6.7 Discussion on Potential Applications

In addition to debugging, the static slices produced by NS-Slicer for incomplete code can be useful in several software engineering applications.

6.7.1 Vulnerability Detection on Incomplete Code Snippets. The ability of NS-Slicer to enable VulDeePecker [Li et al. 2018] to perform vulnerability detection on incomplete code snippets is a significant advancement. This capability facilitates early detection of vulnerabilities, which is crucial in preventing potentially insecure code from being incorporated into software projects. By analyzing incomplete code snippets from sources such as online forums like StackOverflow, NS-Slicer contributes to enhancing the security posture of software projects.

6.7.2 Analyzing Code Snippets from Bug Reports. Developers often encounter defects and report them through issue or bug tracking systems, providing incomplete code snippets where the fault occurred. With NS-Slicer, these snippets can now be analyzed for debugging purposes, enabling developers to better understand and address reported issues. NS-Slicer facilitates a more thorough analysis of code snippets extracted from bug reports. By providing developers with the means to scrutinize incomplete code snippets within the context of reported defects, NS-Slicer empowers them to gain deeper insights into the root causes of issues and devise effective debugging strategies. This capability enhances the efficiency and accuracy of bug resolution processes, ultimately leading to more robust and stable software products.

6.7.3 Code Adaptation. Adapting an incomplete code snippet into a project is a common but challenging task that lacks adequate tool support. NS-Slicer addresses this gap by allowing developers to select only the relevant statements associated with the variable or statement of interest and seamlessly adapt them into their codebases. This capability streamlines the process of integrating code snippets into existing projects. NS-Slicer simplifies the process of adapting incomplete code snippets into existing projects. By allowing developers to selectively choose relevant statements and seamlessly integrate them into their codebases, NS-Slicer streamlines the integration of external code snippets. This feature not only saves developers time and effort but also reduces the likelihood of introducing errors/inconsistencies into the codebase during adaptation.

6.7.4 Compliance Auditing. Incomplete code snippets are often encountered during security audits and compliance checks, particularly when evaluating legacy or third-party code. NS-Slicer plays a vital role in analyzing these snippets to identify potential security and compliance violations, as well as ensuring adherence to coding standards. This application is essential for maintaining a secure and compliant codebase, particularly in regulated industries. Furthermore, NS-Slicer aids in compliance auditing by enabling thorough analysis of incomplete code snippets encountered during security assessments. Its ability to identify potential security and compliance violations, as well as ensure adherence to coding standards, is invaluable for organizations seeking to maintain a secure and compliant codebase. This aspect is particularly critical in industries subject to stringent regulatory requirements, where non-compliance can result in legal and financial consequences.

6.7.5 Analysis on Code Undergoing Edits. During the editing process, code is often incomplete, and existing Integrated Development Environment (IDE) support for such incomplete code is limited to syntactic highlighting. NS-Slicer fills this gap by providing in-depth analysis of program semantics and offering slicing capabilities, empowering developers to better understand and manage incomplete code during the editing process. This enhances the productivity and effectiveness of developers working on evolving codebases.

7 RELATED WORK

7.1 Traditional Program Slicing Approaches

While there is a rich literature in program slicing [Harman et al. 1996; Silva 2012; Tip 1995], none of the existing approaches work for incomplete code snippets. There exist surveys on techniques for program slicing [Binkley and Gallagher 1996; Binkley and Harman 2004; Harman et al. 1996; Harman and Gallagher 1998; Harman and Hierons 2001; Lucia 2001; Tip 1995; Xu et al. 2005]. Silva [Silva 2012] and Harman *et al.* [Harman et al. 1996] provide an extensive survey with multiple dimensions to classify program-slicing works.

NS-Slicer differs from the family of conditioned program slicing [Canfora et al. 1998; Lucia et al. 1996], constraint slicing [Field et al. 1995], and pre/post-conditioned slicing [Harman et al. 2001], where an initial state is defined via conditions.

There are static slicing approaches based on various static analyses, e.g., incremental slicing [Orso et al. 2001], call-mark slicing [Nishimatsu et al. 1999], proposition-based slicing [Hatcliff et al. 2000], stop-list slicing [Gallagher et al. 2006], amorphous slicing [Harman and Danicic 1997]. NS-Slicer is loosely related to SDG-based slicing, as it leverages a SDG-based slicing tool [Galindo et al. 2022] to enable a learning-based approach. But, we do not explicitly build an SDG to compute the slice. There are dynamic slicing approaches [Binkley et al. 2014; Jhala and Majumdar 2005; Korel and Laski 1988; Maras et al. 2011], including language-independent slicing [Binkley et al. 2014], which compute a slice for a specific execution whereas NS-Slicer produces a static slice for all executions.

7.2 Learning-Based Approaches

LExecutor [Souza and Pradel 2023] is a learning-guided approach for executing arbitrary code snippets. It predicts missing values that otherwise would cause the program to get stuck, and to inject these values into the execution. In TRACED [Ding et al. 2024], the authors pre-train code language models with a combination of source code, executable inputs, and corresponding execution traces, and then fine-tune for predicting the execution. The knowledge from both pre-trained language models might benefit dynamic slicing more than static slicing.

NeuralPDA [Yadavally et al. 2023] is a deep learning approach that derives the program dependency graph (PDG) for any complete/incomplete code snippets. However, it cannot be leveraged for program slicing because it operates at the statement level, and does not have have the fine-grained dependencies among program elements in the statements. As a result, the alternative approach of building the PDG for an incomplete code snippet with NeuralPDA, and then performing data/control-flow analysis for extracting the program slice is not possible. In contrast, NS-Slicer works at the token-level and can be leveraged to extract static program slices directly.

8 CONCLUSION

In conclusion, we introduce NS-Slicer, a novel learning-based static program slicing approach which extends the applicability of such techniques to incomplete code. To enable this process, we leverage source code pre-training, and extend it to learn the fine-grained dependencies between the variable at the slicing criterion and all other statements in the program. This knowledge forms the cornerstone while discarding the irrelevant program statements, so as to identify the sets of statements that affect, or are affected by the slicing criterion. In our empirical evaluation, on complete code, NS-Slicer predicts the backward and forward slices with an F1-score of 97.41% and 95.82%, respectively. For partial programs, it records an F1-score of 96.77%–97.49% for backward slicing, 92.14%–95.40% for forward slicing. In addition, we prove the utility of NS-Slicer in detecting vulnerabilities in incomplete code, attaining a prediction F1-score of 73.38%. In effect, we introduce a new semantic knowledge-probing downstream task of static program slicing, proving the efficacy and promise of PLMs in understanding the intricate variable-statement dependencies.

DATA-AVAILABILITY STATEMENT

All of our data, code, and trained models are publicly available on GitHub[2] and Zenodo[3].

ACKNOWLEDGMENTS

This work was supported in part by the US National Science Foundation (NSF) grant CNS-2120386 and the US National Security Agency (NSA) grant NCAE-C-002-2021 on Cybersecurity Research Innovation.

REFERENCES

Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, and William W. Pugh. 2008. Using Static Analysis to Find Bugs. *IEEE Softw.* 25, 5 (2008), 22–29.

David Binkley, Nicolas Gold, Mark Harman, Syed Islam, Jens Krinke, and Shin Yoo. 2014. ORBS: language-independent program slicing. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 109–120. https://doi.org/10.1145/2635868.2635893

David W. Binkley and Keith Brian Gallagher. 1996. Program Slicing. *Adv. Comput.* 43 (1996), 1–50.

David W. Binkley and Mark Harman. 2004. A survey of empirical results on program slicing. *Adv. Comput.* 62 (2004), 105–178.

Gerardo Canfora, Aniello Cimitile, and Andrea De Lucia. 1998. Conditioned program slicing. *Inf. Softw. Technol.* 40, 11-12 (1998), 595–607.

Checkmarx. 2023. *Checkmarx*. https://checkmarx.com/

Kevin Clark, Urvashi Khandelwal, Omer Levy, and Christopher D. Manning. 2019. What Does BERT Look At? An Analysis of BERT's Attention. *CoRR* abs/1906.04341 (2019).

CWE-120. 2023. *CWE-120: Buffer Overflow*. https://cwe.mitre.org/data/definitions/120.html

CWE-290. 2023. *CWE-290: Authentication Bypass by Spoofing*. https://cwe.mitre.org/data/definitions/290.html

CWE-79. 2023. *CWE-79: Cross-site Scripting*. http://cwe.mitre.org/data/definitions/79.html

CWE-89. 2023. *CWE-89: SQL Injection*. https://cwe.mitre.org/data/definitions/89.html

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, 4171–4186. https://doi.org/10.18653/V1/N19-1423

Yangruibo Ding, Benjamin Steenhoek, Kexin Pei, Gail Kaiser, Wei Le, and Baishakhi Ray. 2024. TRACED: Execution-aware Pre-training for Source Code. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering* (, Lisbon, Portugal,) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 36, 12 pages. https://doi.org/10.1145/3597503.3608140

Jiahao Fan, Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories* (Seoul, Republic of Korea) *(MSR '20)*. Association for Computing Machinery, New York, NY, USA, 508–512. https://doi.org/10.1145/3379597.3387501

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *EMNLP (Findings) (Findings of ACL, Vol. EMNLP 2020)*. Association for Computational Linguistics, 1536–1547.

John Field, G. Ramalingam, and Frank Tip. 1995. Parametric program slicing. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '95)*. Association for Computing Machinery, New York, NY, USA, 379–392. https://doi.org/10.1145/199448.199534

Felix Fischer, Konstantin Böttinger, Huang Xiao, Christian Stransky, Yasemin Acar, Michael Backes, and Sascha Fahl. 2017. Stack Overflow Considered Harmful? The Impact of Copy&Paste on Android Application Security. In *Proceedings of the 2017 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 121–136.

[2]https://github.com/aashishyadavally/ns-slicer/

[3]https://zenodo.org/records/10463878

Margaret Ann Francel and Spencer Rugaber. 2001. The value of slicing while debugging. *Sci. Comput. Program.* 40, 2-3 (2001), 151–169.

Carlos Galindo, Sergio Perez, and Josep Silva. 2022. A Program Slicer for Java (Tool Paper). In *Software Engineering and Formal Methods: 20th International Conference, SEFM 2022, Berlin, Germany, September 26–30, 2022, Proceedings* (Berlin, Germany). Springer-Verlag, Berlin, Heidelberg, 146–151. https://doi.org/10.1007/978-3-031-17108-6_9

Keith Gallagher, David Binkley, and Mark Harman. 2006. Stop-List Slicing. In *2006 Sixth IEEE International Workshop on Source Code Analysis and Manipulation*. IEEE Computer Society Press, 11–20. https://doi.org/10.1109/SCAM.2006.30

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie LIU, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCode{BERT}: Pre-training Code Representations with Data Flow. In *International Conference on Learning Representations*. https://openreview.net/forum?id=jLoC4ez43PZ

Daya Guo, Canwen Xu, Nan Duan, Jian Yin, and Julian Mcauley. 2023. LongCoder: A Long-Range Pre-trained Language Model for Code Completion. In *Proceedings of the 40th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 202)*, Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (Eds.). PMLR, 12098–12107. https://proceedings.mlr.press/v202/guo23j.html

Mark Harman and Sebastian Danicic. 1997. Amorphous Program Slicing. In *Proceedings of the 5th International Workshop on Program Comprehension*. IEEE Computer Society, 70–79.

Mark Harman, Sebastian Danicic, Yoga Sivagurunathan, and Dan Simpson. 1996. The Next 700 Slicing Criteria. In *Proceedings of the 2nd U.K. Workshop on Program Comprehension*.

Mark Harman and Keith Brian Gallagher. 1998. Program slicing. *Inf. Softw. Technol.* 40, 11-12 (1998), 577–581.

M. Harman, R. Hierons, C. Fox, S. Danicic, and J. Howroyd. 2001. Pre/post conditioned slicing. In *Proceedings IEEE International Conference on Software Maintenance (ICSM 2001)*. 138–147. https://doi.org/10.1109/ICSM.2001.972724

Mark Harman and Robert M. Hierons. 2001. An overview of program slicing. *Softw. Focus* 2, 3 (2001), 85–92.

John Hatcliff, Matthew B. Dwyer, and Hongjun Zheng. 2000. Slicing Software for Model Construction. *High. Order Symb. Comput.* 13, 4 (2000), 315–353.

José Antonio Hernández López, Martin Weyssow, Jesús Sánchez Cuadrado, and Houari Sahraoui. 2023. AST-Probe: Recovering abstract syntax trees from hidden representations of pre-trained language models. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (, Rochester, MI, USA,) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 11, 11 pages. https://doi.org/10.1145/3551349.3556900

Hyunji Hong, Seunghoon Woo, and Heejo Lee. 2021. Dicos: Discovering Insecure Code Snippets from Stack Overflow Posts by Leveraging User Discussions. In *Proceedings of the 37th Annual Computer Security Applications Conference* (, Virtual Event, USA,) *(ACSAC '21)*. Association for Computing Machinery, New York, NY, USA, 194–206. https://doi.org/10.1145/3485832.3488026

Ranjit Jhala and Rupak Majumdar. 2005. Path slicing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, USA) *(PLDI '05)*. Association for Computing Machinery, New York, NY, USA, 38–47. https://doi.org/10.1145/1065010.1065016

Joern. 2023. *Open-source code analysis platform for C/C++ based on code property graphs.* https://joern.io/

Bogdan Korel and Janusz W. Laski. 1988. Dynamic Program Slicing. *Inf. Process. Lett.* 29, 3 (1988), 155–163.

Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018.* The Internet Society. https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_03A-2_Li_paper.pdf

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. *CoRR* abs/1907.11692 (2019).

Andrea De Lucia. 2001. Program Slicing: Methods and Applications. In *1st IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2001), 10 November 2001, Florence, Italy.* IEEE Computer Society, 144–151. https://doi.org/10.1109/SCAM.2001.972675

Andrea De Lucia, A.R. Fasolino, and M. Munro. 1996. Understanding function behaviors through program slicing. In *WPC '96. 4th Workshop on Program Comprehension*. 9–18. https://doi.org/10.1109/WPC.1996.501116

Josip Maras, Jan Carlson, and Ivica Crnković. 2011. Client-side web application slicing. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. 504–507. https://doi.org/10.1109/ASE.2011.6100110

Georgios Nikitopoulos, Konstantina Dritsa, Panos Louridas, and Dimitris Mitropoulos. 2021. CrossVul: a cross-language vulnerability dataset with commit data. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 1565–1569. https://doi.org/10.1145/3468264.3473122

A. Nishimatsu, M. Jihira, S. Kusumoto, and K. Inoue. 1999. Call-mark slicing: an efficient and economical way of reducing slice. In *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002)*. 422–431.

https://doi.org/10.1145/302405.302674

A. Orso, S. Sinha, and M.J. Harrold. 2001. Incremental slicing based on data-dependences types. In *Proceedings IEEE International Conference on Software Maintenance. ICSM 2001.* 158–167. https://doi.org/10.1109/ICSM.2001.972726

Ruchir Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir R. Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, and Ulrich Finkler. 2021. Project CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks. *CoRR* abs/2105.12655 (2021).

Chaiyong Ragkhitwetsagul, Jens Krinke, Matheus Paixao, Giuseppe Bianco, and Rocco Oliveto. 2021. Toxic Code Snippets on Stack Overflow. *IEEE Transactions on Software Engineering* 47, 3 (2021), 560–581. https://doi.org/10.1109/TSE.2019.2900307

Josep Silva. 2012. A vocabulary of program slicing-based techniques. *ACM Comput. Surv.* 44, 3 (2012), 12:1–12:41.

Beatriz Souza and Michael Pradel. 2023. LExecutor: Learning-Guided Execution. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (, San Francisco, CA, USA,) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 1522–1534. https://doi.org/10.1145/3611643.3616254

Frank Tip. 1995. A survey of program slicing techniques. *J. Program. Lang.* 3, 3 (1995).

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

Morteza Verdi, Ashkan Sami, Jafar Akhondali, Foutse Khomh, Gias Uddin, and Alireza Karami Motlagh. 2022. An Empirical Study of C++ Vulnerabilities in Crowd-Sourced Code Examples. *IEEE Trans. Software Eng.* 48, 5 (2022), 1497–1514.

Yueming Wu, Deqing Zou, Shihan Dou, Wei Yang, Duo Xu, and Hai Jin. 2022. VulCNN: an image-inspired scalable vulnerability detection system. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2365–2376. https://doi.org/10.1145/3510003.3510229

Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. 2005. A brief survey of program slicing. *ACM SIGSOFT Softw. Eng. Notes* 30, 2 (2005), 1–36.

Aashish Yadavally, Tien N. Nguyen, Wenbo Wang, and Shaohua Wang. 2023. (Partial) Program Dependence Learning. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 2501–2513. https://doi.org/10.1109/ICSE48619.2023.00209

Yaqin Zhou and Asankhaya Sharma. 2017. Automated identification of security issues from commit messages and bug reports. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 914–919. https://doi.org/10.1145/3106237.3117771

Received 21-OCT-2023; accepted 2024-02-24