(Partial) Program Dependence Learning

Aashish Yadavally and Tien N. Nguyen
Computer Science Department
The University of Texas at Dallas
Texas, USA
{aashish.yadavally, tien.n.nguyen}@utdallas.edu

Wenbo Wang and Shaohua Wang
Department of Informatics
New Jersey Institute of Technology
New Jersey, USA
ww6@njit.edu, davidshwang@ieee.org

***Abstract*—Code fragments from developer forums often migrate to applications due to the code reuse practice. Owing to the incomplete nature of such programs, analyzing them to early determine the presence of potential vulnerabilities is challenging. In this work, we introduce NeuralPDA, a neural network-based program dependence analysis tool for both complete and partial programs. Our tool efficiently incorporates intra-statement and inter-statement contextual features into statement representations, thereby modeling program dependence analysis as a statement-pair dependence decoding task. In the empirical evaluation, we report that NeuralPDA predicts the CFG and PDG edges in complete Java and C/C++ code with combined F-scores of 94.29% and 92.46%, respectively. The F-score values for partial Java and C/C++ code range from 94.29%–97.17% and 92.46%–96.01%, respectively. We also test the usefulness of the PDGs predicted by NeuralPDA (i.e., PDG*) on the downstream task of method-level vulnerability detection. We discover that the performance of the vulnerability detection tool utilizing PDG* is only 1.1% less than that utilizing the PDGs generated by a program analysis tool. We also report the detection of 14 real-world vulnerable code snippets from StackOverflow by a machine learning-based vulnerability detection tool that employs the PDGs predicted by NeuralPDA for these code snippets.**

***Index Terms*—neural partial program analysis, neural program dependence analysis, neural networks; deep learning**

I. Introduction

Developers often use online question and answering (Q&A) forums, e.g., StackOverflow (S/O), to learn how to use software libraries and frameworks. Sometimes, the answer to a question comes as a fragment/chunk of code, which later makes it to the production applications, stemming from the copy-and-paste software reuse practice. Unfortunately, if the copied code fragments are vulnerable, i.e., possess defects that can potentially be exploited, it will lead to the applications being prone to attacks. Verdi *et al.* [1] reviewed more than 72K C++ code snippets that migrated from 1,325 S/O answers. Of these, they reported a total of 99 vulnerable code snippets of 31 different types that made their way to 2,589 GitHub repositories. Thus, it is crucial to detect early the vulnerabilities in the code snippets from online forums.

Security researchers have proposed several automated approaches for vulnerability detection (VD) using program analysis [2]–[11], as well as machine learning (ML) including deep learning (DL) techniques [12]–[16]. However, these approaches warrant the code to exist as complete program units, often making use of program representations such as Abstract Syntax Tree (AST), Program Dependence Graph (PDG) [12], [16], Control Flow Graph (CFG) [14], Data Flow Graph (DFG) [14], Code Property Graph (CPG) [13], *etc*. At a minimum, they operate at the method-level granularity, making it impossible to utilize them for directly detecting vulnerabilities in code snippets. A possible alternative would be to plug the code snippet into the method, resolve any ambiguities, and test it with a VD tool. However, such a strategy is limited. First, if found vulnerable, the efforts of integrating the code snippet into the existing method would be lost. Second, due to the black-box nature of DL models, we would not know the origin of the vulnerability, i.e., whether it arises due to the flawed code snippet or the existing part of the code.

Besides, analyzing code snippets is not straightforward. Currently, there exist tools such as PPA [17], which parses an incomplete code fragment to extract data types and build an AST in a best-effort manner. StatType [18] resolves the libraries and recovers the fully-qualified names for references. By manual intervention, one can use program analysis tools (e.g., Joern [19]) to derive CFG/PDGs for incomplete code snippets. However, in the code snippets with the statements that contain: (a) references to undeclared libraries, (b) missing variable declarations, (c) unresolved data types, they ignore the edges to/from such statements. Thus, deriving all program dependencies for such incomplete code fragments is not yet possible. Let us call this *partial program dependence analysis*.

In addition to vulnerability detection, such partial program dependence analysis is also beneficial to the other software engineering (SE) tasks that can tolerate a low level of errors and imprecision in building the dependencies. For example, consider code completion [20], [21], in which a model provides suggestions to complete partial code. Existing ML/DL-based code completion models are just based on the code sequences or utilize the syntactic structure in ASTs, but none leverage the program dependencies due to the nature of partial code. Next, consider the task of analyzing the code fragments in a bug report to connect it to the relevant source files for bug localization purposes [22], [23]. Here too, a need for partial program dependence analysis can be observed.

In this paper, we propose NeuralPDA, a neural network-based partial program dependence analysis approach that learns to derive the program dependencies for any code fragments (i.e., both complete and incomplete). We find motivation for such a data-driven, learning-based approach from the following observations. First, in an empirical study on the

repetitiveness, containment, and composability of PDGs in open-source projects, Nguyen *et al.* [24] reported that among 17.5M PDGs with 1.6B PDG subgraphs, 14.3% of the PDGs have all of their subgraphs repeated across different projects. Furthermore, in 15.6% of the PDGs, at least 90% of their subgraphs are likely to have appeared before in other projects. Thus, we design NEURALPDA to learn from PDGs with complete control and data dependencies retrieved from existing code repositories and derive the program dependencies for the (partial) code fragment under study. Second, such a solution is analogous to the neural network-based dependency parsing approaches in natural language processing (NLP), which learn the dependencies signifying the semantic relationships between words in a sentence from text corpora.

NEURALPDA is designed to capture two basic insights about program structure: intra-statement context, and inter-statement context. This is facilitated via a hierarchical, self-attention network (SAN)-based model architecture. The motivation behind intra-statement context learning (IntraS-CL), which aims to learn the context within individual statements, is that IntraS-CL provides the knowledge for a model to learn better the roles and relations among the code tokens in a statement (e.g., definition or usage of a variable), thus, leading to the better discovery of dependencies between two statements. In contrast, inter-statement context learning (InterS-CL) helps a model recognize the dependence of one statement on the other taking into account the surrounding statements.

Assessing the performance of NEURALPDA is not straightforward, mainly due to the lack of ground-truth inter-statement program dependencies for partial code fragments. Thus, we intrinsically evaluated it on programs in Java and C/C++ as follows. First, we trained NEURALPDA on complete code. For testing, we treated each method individually and chose a consecutive portion within the method to predict the program dependencies, and compared them against the actual dependencies. Overall, NEURALPDA predicts CFG and PDG edges in Java with an F-score of 94.29%, and in C++ with an F-score of 92.46%. Upon further investigation, we discovered that for Java, NEURALPDA predicts *sequential* CFG edges, *if-else* CFG edges, *data-dependence* edges, and *control-dependence* edges with an accuracy of 99.4%, 95.52%, 82.78%, and 96.33%, respectively. We also performed an ablation on various model components in NEURALPDA, which enabled us to tie the fine-grained performance gains to each component.

To evaluate the usefulness of the PDGs predicted by NEURALPDA (say, PDG*), we designed experiments around the task of vulnerability detection at two levels of granularity: complete code at the method-level, and partial code at the snippet-level. For the method-level VD task, we leveraged VulCNN [25], an image-inspired DL-based VD model which utilizes PDGs to predict whether a given method has vulnerabilities or not. Here, we aimed to assess how well PDG*s predicted for the methods in the dataset approximate the performance of the actual PDGs retrieved from a program-analysis tool. For this task, VulCNN achieved an F-score of 73.26% with the PDGs predicted by NEURALPDA, as opposed to 74.01% in the case of actual PDGs (1.1% reduction). For the task of partial code VD, we first trained VulCNN on a VD dataset comprising complete C++ methods [26]. Next, we leveraged our tool to predict PDGs for the vulnerable S/O code snippets in Verdi *et al.* [1]. Utilizing these PDGs, VulCNN was able to correctly identify 14 code snippets as vulnerable.

```
1 std :: shared ptr<FILE> pipe(popen(cmd,"r"), pclose);
2 if (! pipe) return "ERROR";
3 char buffer[128];
4 std :: string result = "";
5 while (! feof (pipe.get() ) ) {
6  if ( fgets (buffer , 128, pipe.get() ) != NULL)
7    result += buffer;
8 }
```

Fig. 1: *Execute a command within a C++ program and get output* – Answer ID 478960 on S/O provided in response is prone to OS command injection (CWE-78, CWE-1019) [1].

In brief, this paper makes the following major contributions:

1) NEURALPDA is the first neural network approach to predict program dependencies in complete as well as partial programs, which are accurate as well as 380× faster to generate. This opens up a research direction for improving program analysis (PA) for partial programs by combining our ML/DL approach and top-down PA approaches.
2) An extensive evaluation showing NEURALPDA's high accuracy in deriving program dependencies and its usefulness in the application of vulnerability detection for both complete and incomplete snippets.
3) An analysis showing the usefulness of intra-statement and inter-statement context learning, capturing the higher-order interaction features between statements in a code snippet.

II. MOTIVATION

A. Motivating Examples

The code snippet in Fig. 1 is a response to the question ID 478960 on StackOverflow, seeking to execute a user-input command within a C++ program and retrieve its output. This code is vulnerable to *code injection (OS command injection) attacks*, as the commands input by the user were not validated. For example, it is possible for an attacker to execute privilege-level commands without any errors or warnings. This vulnerability was reported in two Common Weakness Enumeration (CWE): CWE-78 and CWE-1019 (13 different Common Vulnerabilities and Exposures - CVEs).

We can observe that this code snippet is incomplete: (a) variable `cmd` is undeclared; (b) object type `FILE` and the library `std` are undefined; (c) it contains references to functions from external libraries such as `pipe`, `popen`, `pclose`, `feof`, `fgets`, `pipe.get`, etc. On StackOverflow, there is more conversational context to the code snippet which helps developers understand these names and make sense of it. However, neither program analysis nor vulnerability detection tools can analyze this code snippet, resulting in an undetected vulnerability making its way to the programs that directly used this code. Thus, it is

```
1 #include <cstdio>
2 #include <iostream>
3 #include <memory>
4 #include <stdexcept>
5 #include <string>
6 #include <array>
7
8 std::string exec(const char* cmd) {
9   std::array<char, 128> buffer;
10  std::string result;
11  std::unique_ptr<FILE, decltype(&pclose)>
        pipe(popen(cmd, "r"), pclose);
12  if (!pipe) {
13    throw std::runtime_error("popen() failed!");
14  }
15  while (fgets(buffer.data(), buffer.size(), pipe.get())
        != nullptr) {
16    result += buffer.data();
17  }
18  return result;
19 }
```

Fig. 2: Complete Code with same POSIX elements as Fig. 1

desirable to have an automated analysis tool that can analyze incomplete code. PPA [17] can parse a code fragment to build an AST and extract data types in a best-effort manner, and StatType [18] can derive the fully-qualified names for references. However, none of the state-of-the-art approaches can retrieve the inter-statement *program dependencies* in incomplete code, which is crucial in understanding/analyzing the vulnerabilities in a code snippet. With manual intervention, one could use program analysis tools (e.g., Joern [19]) to derive CFGs/PDGs on such an incomplete code snippet. However, those tools will ignore the dependencies to and from the statements with undeclared variables (`cmd`), undefined types (`FILE`), and undeclared libraries (`std`). Let us refer to such an analysis as *partial program dependence analysis*.

Aside from detecting vulnerabilities in incomplete code, such partial program dependence analysis (Partial PDA) is beneficial to automated code completion (CC) tools as well. For example, in Fig. 1, assume that a developer editing the code invokes the CC tool at line 5: `! feof (pipe._`. With the program dependencies retrieved from partial PDA, the CC tool could suggest `pipe.get` due to the knowledge of a control dependency between the statement `pipe(popen(...))` on line 1 and the potentially suggested candidate `pipe.get`.

Observation 1 (Partial Program Dependence Analysis). *Partial program dependence analysis is desirable for the tasks in which the completely analyzable code is unavailable. Such an analysis is useful for tasks that can tolerate a low level of errors and imprecision in deriving those dependencies*

Now, consider the complete code example in Fig. 2, from S/O post 10702464 [27]. While there are slight differences in a few details (constants, error messages, etc.) between the incomplete code snippet in Fig. 1 and the complete code example in Fig. 2, the presence of many similar statements indicates that the data and control dependencies are comparable: line 1 (Fig. 1) and line 11 (Fig. 2), line 2 (Fig. 1) and lines 12–14 (Fig. 2), lines 3–4 (Fig. 1) and lines 9–10 (Fig. 2), lines 5–6 (Fig. 1) and line 15 (Fig. 2), line 7 (Fig. 1) and line 16 (Fig. 2). Thus, the program dependencies between the statements in the incomplete code snippet in Fig. 1 can be learned from those extracted for the complete code example in Fig. 2.

Observation 2 (Learn to Analyze Program Dependencies). *Finding patterns from complete code in existing code corpora could be a good strategy to learn to analyze the inter-statement program dependencies in a given incomplete code snippet.*

B. Key Ideas

Following Observations 1–2, we design NeuralPDA for partial program dependence analysis with the following key ideas:

1) **[Key Idea 1] Neural Network-Based Approach to Partial Program Dependence Analysis***:* Instead of deterministically producing the program dependencies in a best-effort manner, following Observation 2, we design a deep learning model (DL) to learn to analyze the program dependencies among the statements in the given source code. By leveraging the program dependencies extracted by program analysis techniques [28] for the complete code in the open-source projects (e.g., GitHub) in the training process, the DL model can derive the inter-statement program dependencies for a given code snippet.

2) **[Key Idea 2] Program Dependence Decoding from Dense Statement Representations***:* We seek inspiration from the neural network-based dependency parsing approaches [29] in NLP. They successfully learn the semantic relations between the words in a sentence by learning the dependencies between their latent representations obtained from mapping them into an embedding space. Following suit, we design NeuralPDA to learn the representations for the statements in source code so as to learn the program dependence relations between them.

3) **[Key Idea 3] Enhancing Statement Representations with Intra-Statement and Inter-Statement Context Learning***:* The quality of the statement representations determines how accurately NeuralPDA can predict the program dependence relations. To this effect, we rely on two types of contextualization. **Intra-statement context** refers to the program entities represented by the code tokens within an individual statement, which helps the model derive the control/data dependencies among the statements. For example, in Fig. 1, the variable declaration statement on line 3 contains the token (variable) `buffer`, which is also referred to in the assignment statement on line 7. Intra-statement contextualization makes information about the local context within the individual statements on line 3 and line 7 available globally, thus helping the model recognize the declaration and reference of the same variable. This facilitates the recognition of the data dependency between the two statements via a def-use relation with variable `buffer`.

For the two statements under study, **inter-statement context** helps NeuralPDA model the effect that all the other statements in the code snippet have on the relationship between them. For example, in Fig. 1, knowing that the `if` statement on line 6 is nested within a `while` loop on lines 5–8 will help

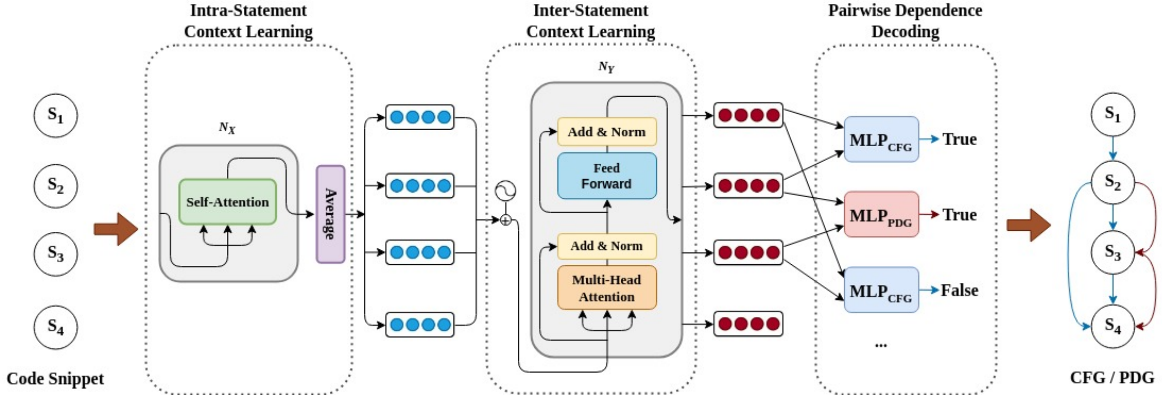


Fig. 3: NEURALPDA: Model Architecture

the model recognize that the execution of assignment statement on line 7 depends not only on the condition on line 6, but also on the loop-condition on line 5.

III. NEURALPDA: MODEL OVERVIEW

Given a code snippet covering the statements s_1, s_2,.., s_N, where the statement s_i contains the tokens $t_1^{(i)}$, $t_2^{(i)}$.., $t_M^{(i)}$, as illustrated in Fig. 3, NEURALPDA has the following essential components to learn the program dependencies between the statement pairs $<s_i, s_j>$ (where $1 \leq i, j \leq N$):

A. Intra-Statement Context Learning

For the tokens $t_1^{(i)}$, $t_2^{(i)}$.., $t_M^{(i)}$ in the statement s_i, the goal of this component is to map them into an embedding space $\mathbb{R}^d$, and generate a context-dependent representation $u_i \in \mathbb{R}^d$ for statement s_i. Such an intra-statement contextualization helps NEURALPDA model the syntactic and semantic relationships between the individual tokens within the context of an individual statement and relay this knowledge to the other statements in the code snippet. We enable this via a position-encoded, simple (i.e., $N_X = 1$) self-attention network (SAN).

B. Inter-Statement Context Learning

Given statement representations $u_i \in \mathbb{R}^d$ for the statements s_1, s_2,.., s_N in a code snippet that are local context-aware, the goal of this component is to learn their latent vector representations $v_i \in \mathbb{R}^d$ that model the inter-statement context including the dependencies between the statements. Such an inter-statement contextualization helps NEURALPDA learn the important dependencies between the statements in the context of the surrounding statements. We enable this via a multi-layer (N_Y=6) bidirectional Transformer encoder [30].

C. Pairwise Dependence Decoding

Given intra-statement and inter-statement contextualized vector representations $v_i \in \mathbb{R}^d$ for the statements s_1, s_2,.., s_N in a code snippet, the goal of this component is to score program dependence edges and control-flow edges for all the statement pairs $<s_i, s_j>$, a combination of which can be formalized as the directed graphs in CFG/PDG. We enable such an edge-scoring approach by using two multi-layer perceptrons (MLP), one corresponding to the control-flow graph (MLP_{CFG}), and the other corresponding to the program dependence graph (MLP_{PDG}).

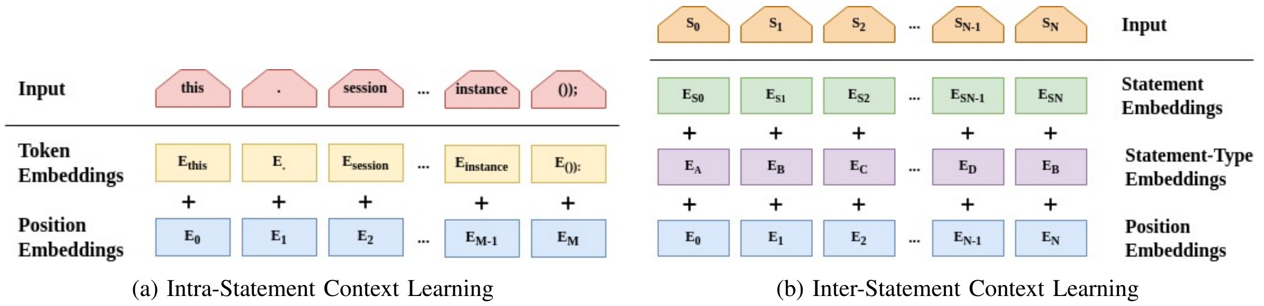
IV. NEURAL PROGRAM DEPENDENCY ANALYSIS

In this section, we will present our model's architecture, training and inference processes to predict CFG/PDGs for complete and partial programs.

A. Model Architecture

As explained in Section III, better contextualization is the main idea behind NEURALPDA's design. Given that attention is the component in the ubiquitous Transformers' [30] success in efficiently learning representations for different entities in different contexts, we chose to make it the foundation of our model. In brief, we realize NEURALPDA via a hierarchical, self-attention network (SAN)-based model architecture, where each sub-network is intended to capture different aspects of contextualization. Its details are as follows:

*1) **Intra-Statement Context Learning (IntraS-CL)**:* The syntactic and semantic knowledge of the code tokens within a single statement must be made available globally to other statements in a program to learn the inter-statement program dependencies effectively. We enable this via a *1-Layer* (i.e., N_X=1) Self Attention Network (1L-SAN). The self-attention layer in an 1L-SAN inputs $x_1, x_2, ..., x_n \in \mathbb{R}^d$, performs self-attention once by projecting the inputs from all attention heads $\in \mathbb{R}^{d_h}$ into the head dimension space d_h via linear transformations, and generate outputs $y_1, y_2, ..., y_n \in \mathbb{R}^d$ which are linear combinations of the concatenated attention head values. We use one attention head (i.e., h=1) for the self-attention layer in 1L-SAN, and the size of the input representations, i.e., d is set to 512. Our experiments revealed only a marginal performance gain by expanding the 1L-SAN to a 2L-SAN, which also came with high computational overhead. Besides, increasing the number of attention heads did not help either performance or interpretability. A more detailed analysis on hyper-parameters and subsequent trade-offs is left to future work.



(a) Intra-Statement Context Learning

(b) Inter-Statement Context Learning

Fig. 4: Input representations of (a) tokens in a statement are the sums of token embeddings, and (token) position embeddings; (b) statements in a snippet are the sums of statement embeddings, statement-type embeddings, and (statement) position embeddings.

Token Input Representations. For a program with N statements $s_1, s_2, ..., s_N$, NEURALPDA takes as input a concatenation of N sequences of M tokens each, $\langle t_1^{(1)}, t_2^{(1)} .., t_M^{(1)} \rangle$, ..., $\langle t_1^{(N)}, t_2^{(N)} .., t_M^{(N)} \rangle$. Next, each token sequence $\langle t_1^{(i)}, t_2^{(i)} .., t_M^{(i)} \rangle$ is input to the 1L-SAN for intra-statement contextualization. Previous works [31], [32] have demonstrated the advantages of a byte-level Byte-Pair Encoding (BPE)-scheme for tokenization. We follow suit to train a byte-level BPE tokenizer for converting a given statement into a sequence of tokens. Here, M is the maximum number of tokens allowed in a statement. For statements with token sequences having less than M tokens, a special *[PAD]* token is appended. In contrast, token sequences having $>M$ tokens are truncated to M tokens.

Token Embeddings. For all the words in the vocabulary V, we leverage a learnable embedding to learn and store their representations (i.e., $\mathbb{R}^{|V| \times d}$). Using this as a lookup table, token embeddings are retrieved for all the tokens generated by the tokenizer for a given statement.

Token Position Embeddings. Attention mechanism in the self-attention layer is invariant to position information. However, this knowledge is key to understanding the sequential nature of code tokens in a statement. We enable this via learnable position encoding scheme, where a vector $\in \mathbb{R}^d$ unique to each position is learned during the training process.

Statement (Output) Representations. Input representations to the 1L-SAN corresponding to the tokens in a given statement are taken as the sums of the *token embeddings*, and their *position embeddings* (as in Fig. 4a). The 1L-SAN yields intra-statement contextualized token representations as its output, which are then averaged to retrieve the statement representation. Note that the token representations corresponding to the *[PAD]* tokens are not considered for averaging. Such statement representations u_i ($\in \mathbb{R}^d$) for all the statements $s_i \in s_1 ... s_N$ are then passed on for inter-statement contextualization.

2) **Inter-Statement Context Learning (InterS-CL):** The knowledge of surrounding statements in the context of a given statement helps NEURALPDA model the dependencies between them better. We enable this via a multi-layer bidirectional Transformer encoder based on the work by Vaswani *et al.* [30]. Owing to its common usage, we will omit exhaustive background details on Transformers' model architecture and will refer the readers to its paper [30]. As shown in Fig. 3, we denote the number of layers in the Transformer encoder by N_Y, which we set to 6. We employ 4 attention heads, i.e., h=4 to increase parallelization (since $d_h = \frac{d}{h}$, i.e., d_h=128) and learn different aspects of the syntactic and semantic structure in the statements, while still being interpretable. We also set the feed-forward module size to be 4 times that of the size of the input representations d, i.e., 2048. Overall, the Transformer encoder in InterS-CL phase inputs local context-aware statement representations $u_i \in \mathbb{R}^d$ for all statements s_i in a given program, and outputs statement representations $v_i \in \mathbb{R}^d$ that are both local and global context-aware.

Statement Input Representations. For all the statements s_i in a program, statement embeddings (i.e., $u_i \in \mathbb{R}^d$) are obtained from the 1L-SAN in the IntraS-CL phase. N is the maximum number of statements in a method in the dataset. If a given program has less than N statements, zero vectors ($\in \mathbb{R}^d$) are padded to the inputs.

Statement Position Embeddings. To make our model understand the sequential nature of statements in a program, as in Section IV-A1, we leverage a learnable position encoding scheme to learn unique vectors ($\in \mathbb{R}^d$) for all statement positions.

Statement Types. Most neural network-based dependency parsers leverage parts-of-speech (POS) tags for the words in a sentence for better dependency learning. Thus inspired, we chose to associate with each statement a label indicating the *statement type*, which is essentially the type of the AST node rooted at the sub-AST for the statement. We extract labels such as `METHOD`, `CONTROL_STRUCTURE`, `BLOCK`, etc., which helps augment the statement with such syntactic information. We learn unique vectors ($\in \mathbb{R}^d$) for each of the statement types.

Statement (Output) Representations. As shown in Fig. 4b, the input representations for the statements in a program are taken as the sums of the *statement embeddings*, *statement-type embeddings*, and the *statement position embeddings*. These are then passed on to the Transformer encoder, to retrieve contextualized statement representations $v_i \in \mathbb{R}^d$ for all the statements $s_i \in s_1 ... s_N$, that model the syntactic and semantic knowledge from both within and across the statements. After this, we obtain the contextualized statement representations.

3) **Pairwise Dependence Decoding:** From the sequence of contextualized statement representations $v_i \in \mathbb{R}^d$ corresponding to all the statements in a program passed on by the Transformer encoder in the InterS-CL phase, pairs such as $\langle v_i, v_j \rangle$ (1≤ i, j ≤ N) are taken to detect the presence of CFG/PDG edges between the two statements s_i and s_j. We leverage 2-layered multi-layer perceptron networks (each for detecting the CFG and PDG edges, i.e., MLP_{CFG} and MLP_{PDG}, respectively) in the pairwise dependence decoding phase, which are scored as follows:

$$score_{\text{rel}}(i,j) = MLP_{\text{rel}}(v_i \circ v_j \circ (v_i * v_j) \circ |v_i - v_j|) \quad (1)$$

where $\circ$, $*$ and $|.|$ correspond to concatenation, element-wise product, and absolute element-wise difference operations respectively; and *rel* represents either the control-flow or program dependence relations. Attaining a $score_{\text{rel}}(i,j) > 0.5$ represents the detection of the corresponding CFG/PDG edge from the statement s_i to the statement s_j. The combination of all the CFG/PDG edges extracted via such an arc-factored approach is realized as the CFG/PDG for the given program.

B. Training Process

Training NEURALPDA requires the knowledge of ground-truth CFG and PDG edges between program statements. Thus, it can only be trained on complete programs (at a minimum, which are at a method-level granularity) so as to be able to leverage program analysis tools to extract them. The CFG and PDG edge information can then be utilized to compute the training objective loss (i.e., $\mathcal{L}$) for our model as follows:

$$\mathcal{L} = \mathcal{L}_{\text{CFG}} + \mathcal{L}_{\text{PDG}} \quad (2)$$

where $\mathcal{L}_{\text{CFG}}$ is the loss for CFG edge-decoding, and $\mathcal{L}_{\text{PDG}}$ is the loss for PDG edge-decoding. Moreover, $\mathcal{L}_{\text{CFG}}$ and $\mathcal{L}_{\text{PDG}}$ are computed as the sums of all binary-cross entropy (BCE) losses corresponding to the CFG and PDG edge predictions between different statements in a program. Note that the inter-statement losses corresponding to the edges from/to the zero-padded statements do not contribute to either $\mathcal{L}_{\text{CFG}}$ or $\mathcal{L}_{\text{PDG}}$. The model parameters which are learned to minimize $\mathcal{L}$ include learnable embeddings (token, token position, statement type, and statement position), attention, Tr-FFNN (i.e., feed-forward neural network in Transformer encoder), MLP_{CFG}, and MLP_{PDG}. Overall, NEURALPDA has about 39M parameters.

C. Inference for Dependency Discovery

Despite being trained on only complete code, one can leverage NEURALPDA to extract the control-flow and program dependence edges for both complete and partial code. The following, however, are the important points of consideration:

- *Statement Types*: To extract the syntactic information encoded in *statement types*, one would need the program's AST. In Java, for example, this can be retrieved even if the code is incomplete using tools such as PPA [17]. However, this is not possible for all programming languages. In such cases, NEURALPDA can be trained without statement types, i.e., by computing the input representation for statements in a program as just the sums of the statement and their position embeddings. In Section VI-D, we demonstrate the practicality of such an alternative.
- *Programs with* $\leq N$ *statements*: Making use of a trained NEURALPDA model on both complete and partial programs, the number of statements in which is less than the *maximum statements* allowed in the model is straightforward. In such cases, NEURALPDA predicts the CFG/PDG edges from one statement to another by contextualizing over all the other statements in the program.
- *Programs with* $>N$ *statements*: For (both complete and partial) programs having number of statements greater than that allowed in the trained NEURALPDA model, we have the following strategies in NEURALPDA: (a) train a model with a higher value of N, (b) chunk the program into N-statement code fragments, predict CFG/PDG edges for each of the code fragments independently, and finally, combine the CFG/PDG predictions for all the fragments. For example, if a trained model allows a maximum of 16 statements, to predict for a program with 46 statements, one can break it down into code fragments with 16, 16, and 14 statements, respectively. A potential downside to strategy (b), however, is that a statement in a fragment will be contextualized only over the other statements in that fragment, and the control-flow and program dependencies across fragments will not be captured. Increasing N could address this issue, albeit with more computational overhead.

V. EMPIRICAL EVALUATION

To fully evaluate NEURALPDA, we design a series of experiments seeking to answer the following questions:

(RQ_1) Effectiveness Evaluation on Java code

1.1 *Intrinsic Evaluation*. How accurate is NEURALPDA in generating CFG/PDGs for complete/partial Java code?

1.2 *Qualitative Evaluation*. With what accuracy does NEURALPDA predict specific program dependence relations?

1.3 *Ablation Study*. How do the different components in NEURALPDA contribute to its model performance?

(RQ_2) Effectiveness Evaluation on C/C++ code

How accurate is NEURALPDA *in generating CFG/PDGs for C/C++ code? How about specific dependency relations?*

(RQ_3) Vulnerability Detection in Complete Methods

How do the predicted PDGs from NEURALPDA *approximate the extrinsic task of vulnerability detection on complete code?*

(RQ_4) Vulnerability Detection in StackOverflow Snippets

How do the predicted PDGs from NEURALPDA *help with extrinsic task of discovering vulnerabilities in S/O code snippets?*

VI. EVALUATING EFFECTIVENESS FOR JAVA AND C/C++

A. Data Collection, Procedure, and Evaluation Metrics

To enable the effectiveness evaluation (RQ_1–RQ_2), we collected and filtered Java and C/C++ data in the following way:

- *Java*. GitHub Java Corpus [33] is a large-scale collection of Java code containing 10,968 training and 3,817 testing projects. Within each project, we retained only the source

TABLE I: Effectiveness on Complete Methods in Java ($RQ_{1.1}$)

P/L	Graph	Accuracy	Precision	Recall	F-Score
Java	*CFG*	99.79	98.31	98.58	98.44
	PDG	98.87	89.89	87.53	88.70
	Overall	99.33	94.75	93.83	**94.29**

code files. Each file has from 100 to 512 tokens. We then collapsed the directory structure within the projects and randomly selected 57,600 Java files from the projects in the training set, and 14,400 from the ones in the testing set.

- *C/C++*. Fan *et al.* [26] collected *Big-Vul*, a large C/C++ vulnerability dataset from the Common Vulnerabilities and Exposure database. We further expanded this dataset to span across 2000–2021 and extracted ∼50K methods from it.

Next, we leveraged the Joern program analysis tool [19] to extract the AST edges (to retrieve syntactic type information), CFG edges, and PDG edges for the files in the Java dataset, and the methods in the C/C++ dataset. We also filtered out the Java and C/C++ methods without any CFG or PDG edges.

Finally, we split both Java and C/C++ datasets into: (a) 40,000 Java methods for training, and 4,000 each for validation and testing; (b) 12,767 C/C++ methods for training, and 1,500 each for validation and testing. With an initial learning rate of $5\text{x}10^{-4}$, and by setting the maximum number of tokens in a statement (i.e., M) to 32, and the maximum number of statements in a program (i.e., N) to 8, we trained NEURALPDA on both datasets. The training process was carried out on a machine with an Nvidia Quadro P4000 GPU. The training time per epoch was ∼4.5 hours and ∼1.5 hours for Java and C/C++ datasets, respectively (see Section X).

We model the program dependence decoding step in NEURALPDA as a classification problem. Thus, we adopt the standard evaluation metrics: Accuracy = $\frac{TP+TN}{TP+FP+FN+TN}$, Recall = $\frac{TP}{TP+FN}$, Precision = $\frac{TP}{TP+FP}$, and F-Score = $\frac{2*Recall*Precision}{Recall+Precision}$. Here, TP = True Positives, FP = False Positives, FN = False Negatives, and TN = True Negatives.

B. Effectiveness on Java code ($RQ_{1.1}$)

1) Effectiveness on Complete Java Code: Table I shows the performance of NEURALPDA on the test set comprising the complete methods from the Java dataset. Clearly, our model produces competitive results. In particular, NEURALPDA predicts the control-flow edges with an F-score of 98.44% and the program-dependence edges with an F-score of 88.70%. Overall, it approximates the combination, i.e., CFG + PDG generated by the program analysis tool for the complete methods with an F-score of 94.29%.

2) Effectiveness on Partial Java Code: To show NEURALPDA's flexibility in handling both complete and partial code, we evaluated our model's performance on consecutive lines of code in a method, starting from the first statement. We refer to this as the Top-N experimental setting. In this experiment setting with N ranging from 3 to 8, if a method has $n \leq 8$ statements, it is treated as a partial method for N from $3 \rightarrow n$, and as a complete method for N from $n \rightarrow 8$.

TABLE II: Effectiveness on Partial Code in Java ($RQ_{1.1}$)

N	F-Score		
	CFG	*PDG*	*Overall*
3	99.70	93.24	97.17
4	99.41	92.51	96.61
5	99.26	91.23	95.96
6	99.04	90.36	95.40
7	98.75	89.45	94.82
8	98.44	88.70	**94.29**

TABLE III: NEURALPDA's Performance for different types of Control-Flow and Program Dependence Edges in Java ($RQ_{1.2}$)

Graph	Edge Type	%C
CFG	*sequential*	99.54
	if-else	95.52
PDG	*data dependence*	82.78
	control dependence	96.33

However, in cases where a method has $n > 8$ statements, it is treated as a partial method for all values of N. As seen in Table II, the overall F-score decreases gracefully as the size of partial code fragment increases because more edges need to be predicted. Also, F-score decreases only <3% as N increases from $3 \rightarrow 8$. Thus, NEURALPDA is effective for partial code.

NEURALPDA approximates the combined CFG + PDG for Java code with an overall F-score of 94.29 – 97.17%.

C. Qualitative Analysis of NEURALPDA *for Java ($RQ_{1.2}$)*

To better understand the model performance, we analyzed how precisely NEURALPDA can predict different types of control-flow and program dependence edges in the Java dataset. We enable this by retrieving two kinds of control-flow edges: *sequential* and *if-else*. Here, we identify all edges whose control flows from an `if`-statement to an `else`-statement as an *if-else* edge. In addition, we retrieve two kinds of program dependence edges: *data* and *control dependence*.

In Table III, we report the results for our qualitative evaluation, where %C corresponds to the percentage of correctly predicted edge relations. Overall, NEURALPDA predicts the *sequential*, *if-else*, *data dependence*, and *control dependence* edges with an accuracy of 99.54%, 95.52%, 82.78%, and 96.33% respectively. The relatively lower accuracy in predicting data dependence edges can possibly be attributed to a class-imbalance problem since the percentage of the statement pairs having data dependencies is much smaller than that of pairs without them. This can be boosted by: (a) expanding the dataset to contain more data dependence edges, (b) incorporating class-imbalance mitigation strategies.

NEURALPDA predicts the *sequential* and *if-else* CFG edges with an F-score of 99.54% and 95.52% respectively; *data* and *control* dependence PDG edges with an F-score of 82.78% and 96.33% respectively.

TABLE IV: Ablation over Model Components ($RQ_{1.3}$)

Baselines	Accuracy	Precision	Recall	F-Score
(a) w/o Statement PE	97.10	82.26	64.61	72.37
(b) w/o Statement Types	99.15	93.16	92.39	92.78
(c) w/o IntraS-CL	97.77	89.49	70.29	78.74
(d) w/o InterS-CL	98.53	89.89	84.56	87.14
NEURALPDA	99.33	94.75	93.83	**94.29**

TABLE V: Qualitative Evaluation of LOO-NEURALPDA

Baselines	CFG		PDG	
	sequential	*if-else*	*data*	*control*
(a) w/o Statement PE	55.30	68.66	58.39	63.63
(b) w/o Statement Types	98.31	92.54	81.63	95.78
(c) w/o IntraS-CL	93.13	31.34	43.60	47.50
(d) w/o InterS-CL	99.02	79.10	84.19	95.41

D. Ablation Study and Analysis ($RQ_{1.3}$)

We performed an ablation over different components in NEURALPDA to better understand their relative importance. We refer to such an ablation setting as Leave-One-Out (LOO)-NEURALPDA in which we built different baselines by removing one of the components. These include:

1) W/o Statement Position Embeddings (PE): In Section IV-A, we hypothesized that the knowledge of statement positions is crucial to learn the sequential nature of source code. Thus, we created this baseline to evaluate the importance of statement positions. Here, the input representations for the statements in a code snippet are computed as just the sums of the statement embeddings and the statement-type embeddings.

2) W/o Statement Types: To better learn the syntactic nature of a statement, we hypothesized the inclusion of statement types. In this baseline, we assess their importance by computing the input representations for the statements in a snippet as just the sums of the statement and their position embeddings.

3) W/o IntraS-CL: We created this baseline to better evaluate the importance of *intra-statement context learning*. Instead of using a self-attention network (Fig. 3) to compute the statement representations, we use Word2Vec [34] to compute the word embeddings for each token in the statement and consider their average as the statement representation.

4) W/o InterS-CL: We created this baseline to better evaluate the importance of *inter-statement context learning*. The statement representations generated by IntraS-CL (Fig. 3) for all the statements in a code snippet are directly input to the pairwise-decoder MLPs, instead of being passed to the Transformer encoder for inter-statement contextualization.

Results: As shown in Table IV, our design choices help achieve higher overall accuracy than all LOO-NEURALPDA baselines. In particular, our model improves over the baselines (a)–(d) by over 30.29%, 1.63%, 19.75% and 8.21% respectively. Furthermore, it can be seen that NEURALPDA w/o statement types achieves the performance closest to that of NEURALPDA. This is particularly useful when leveraging it to derive the CFG and PDG edges for code snippets in which *extracting such syntactic type information is not possible*.

TABLE VI: Effectiveness on Complete C/C++ Methods (RQ_2)

P/L	Graph	Accuracy	Precision	Recall	F-Score
C/C++	*CFG*	99.50	96.76	96.56	96.66
	PDG	98.55	83.55	90.01	86.66
	Overall	99.02	91.10	93.87	**92.46**

TABLE VII: Effectiveness on Partial Code in C/C++ (RQ_2)

N	F-Score		
	CFG	*PDG*	*Overall*
3	98.39	91.10	96.01
4	98.33	90.18	95.28
5	97.91	89.10	94.37
6	97.14	87.91	93.31
7	96.69	86.45	92.39
8	96.66	86.66	**92.46**

TABLE VIII: Model Performance for different Types of Control-Flow and Program Dependence edges in C/C++ (RQ_2)

Graph	Edge Type	%C
CFG	*sequential*	98.91
	if-else	**
PDG	*data dependence*	88.21
	control dependence	94.65

Ablation Baselines and Different Types of Dependencies: We also extend the qualitative analysis (as in Table III) to the LOO-NEURALPDA baselines, to investigate how the absence of different model components contributes to the imprecision in predicting different kinds of control-flow and program dependence edges. From Table V, it can be seen that:

- In the absence of the *position information* of statements, the ability of NEURALPDA to predict all CFG and PDG edges drops significantly, which is justified. For example, without such knowledge, the model might not know that the control should always flow from an `if`-statement to its corresponding `else` statement, or that the direction of the data dependence should be from a variable definition to its reference in a `def-use` chain, etc.
- Statement types like *call*, *return*, etc. help the model capture the control dependencies better. Thus, in the absence of such syntactic information, the drop in F-score in predicting the control-flow and control-dependent edges is expected.
- In Key Idea 3 (see Section II-B), we establish the importance of intra-statement context (IntraS-CL) in determining a data dependency between two statements. The drop in F-score in the prediction of data-dependence edges by 89.86% in the absence of such contextualization corroborates that claim.

All model components in NEURALPDA directly contribute to its ability in predicting different types of control-flow and program dependence relations.

E. Effectiveness on C/C++ code (RQ_2)

1) Effectiveness on Complete C/C++ Code: In Table VI, we report NEURALPDA's performance on test set compris-

ing complete methods from the C/C++ dataset. Overall, our model approximates the combination, i.e., CFG+PDG generated by the program analysis tool for the complete methods with 92.46% in F-score, which is consistent with that for Java.

2) Effectiveness on Partial C/C++ Code: Next, as in Section VI-B2, we extend the Top-N experimental setting to the C/C++ methods as well (Table VII). As N increases from 3→8, F-score decreases by only <4%, thus showing NeuralPDA's efficacy in handling partial C/C++ code.

3) Qualitative Analysis of Effectiveness on Different Types of Dependencies for C/C++: We conducted a qualitative analysis to test our tool's effectiveness in predicting different kinds of control-flow and program dependence edges. As seen in Table VIII, it predicts *sequential*, *data dependence*, and *control dependence* edges with an accuracy of 98.91%, 88.21%, and 94.65%, respectively. None of the C/C++ methods in test set have *if-else* CFG edges, as indicated by "**". However, note the increase in the prediction of data dependence edges (i.e., 88.21%) from 82.78% for the Java methods. That increase can be seen despite a significant decrease in the sizes of the Java and C/C++ datasets (∼48K in Java to ∼15K in C/C++). This further confirms that the ability of NeuralPDA to predict different types of CFG and PDG edges is directly dependent on the distribution of the relations in the dataset.

> The efficacy of NeuralPDA is programming language-agnostic, as it approximates the combined CFG+PDG for C/C++ methods with an F-score of 92.46 – 96.01%.

VII. What does NeuralPDA learn? A Case Study

Consider the source code example illustrated in Fig. 5 (top), which was retrieved from the test-split in the Java dataset (see Section VI-A). We predict PDGs for this Java method by employing three NeuralPDA variants: (a) *baseline*; (b) *w/o IntraS-CL*, i.e., by dropping the 1L-SAN; (c) *w/o InterS-CL*, i.e., by dropping the Transformer encoder. The PDGs so predicted are depicted in subfigures (a)–(c) in Fig. 5.

As seen in Fig. 5(a), NeuralPDA predicts all CFG/PDG edges accurately and misses one control-dependence edge $S_2 \rightarrow S_7$. However, given that the baseline correctly identifies the control-flow edge $S_2 \rightarrow S_7$, it is possible that it could not conclude whether S_2 determines the execution of S_7, resulting in the miss. In contrast, the variant *w/o IntraS-CL* misses a data-dependent edge $S_1 \rightarrow S_2$, and a control-dependent edge $S_2 \rightarrow S_3$. This can be due to how statement representations are computed in this variant (see Section IV-A1), causing the model to lose the sense of data propagation. Interestingly, this variant captures a control-dependence edge $S_2 \rightarrow S_7$, while not capturing any control-flow between them – thus appearing to be less intelligent. This further verifies our hypothesis in Key Idea 3, that IntraS-CL component is essential to relay the intrinsic knowledge within a statement globally.

Next, in Fig. 5(c), we can see that the variant *w/o InterS-CL* misses both control-flow and control-dependent edges from S_2 to S_7 and incorrectly predicts (false positive) both of

```
S1  private boolean isValidUntil(Until annotation) {
S2  if (annotation != null) {
S3    double annotationVersion = annotation.value();
S4    if (annotationVersion <= version) {
S5      return false; }
S6    }
S7    return true;
S8  }
```

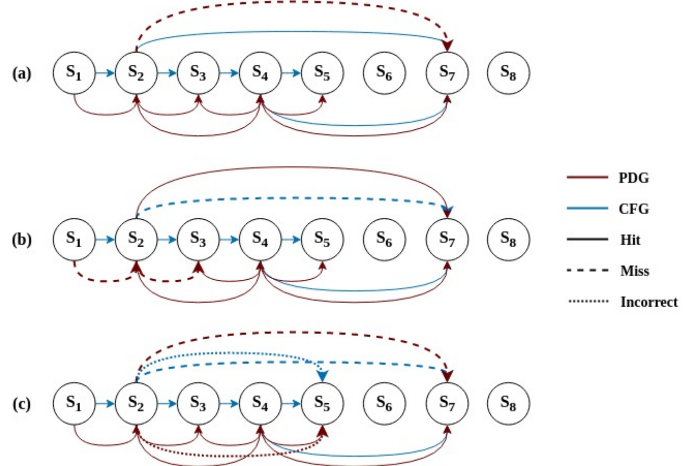


Fig. 5: CFG/PDGs predicted for a Java program by: (a) NeuralPDA, (b) *w/o IntraS-CL*, (c) *w/o InterS-CL*.

these from S_2 to S_5. In addition, NeuralPDA's performance drops from 95.52% (in Table III) to 79.10% (w/o InterS-CL in Table V) in the case of *if-else* relations, prompting us to reckon it is a general trend. Thus, we can corroborate Key Idea 3 by demonstrating the inadequacy in understanding program semantics without inter-statement contextualization.

Previous works [30], [35] have shown that the attention heads in multi-layer SAN-based models tend to capture specific syntactic and semantic properties. To better interpret what NeuralPDA learns in this example, we plotted the attention maps for all the heads across six layers in our model's InterS-CL component, some of which we illustrate in Fig. 6. In Fig. 6(a)–(c), a higher dependence of S_i on S_j is indicated by a darker edge. For example, in Fig. 6(a), we can see that in the 4th attention head on the 6th layer (L6-H4), S_4 attends to S_1, S_2, S_3, S_4, and S_7. More interestingly, in Fig. 6(b), i.e., in L4-H1, all the statements attend to S_5 but not S_7 (indicated in red). Similarly, in Fig. 6(c), i.e., in L3-H3, we can see that all the statements attend to S_7 but not S_5. NeuralPDA possibly learns the sense of an execution trace on these attention heads, because if S_5 is executed, S_7 cannot be, and vice versa. Refer to our webpage [36] for all other attention map illustrations.

VIII. Method-Level Vulnerability Detection

Deep learning (DL)-based approaches that utilize PDGs for vulnerability detection (VD) can tolerate a low level of errors in the program dependencies, wherein the imprecision acts as noise and aids in regularizing the model. VulCNN [25] is one

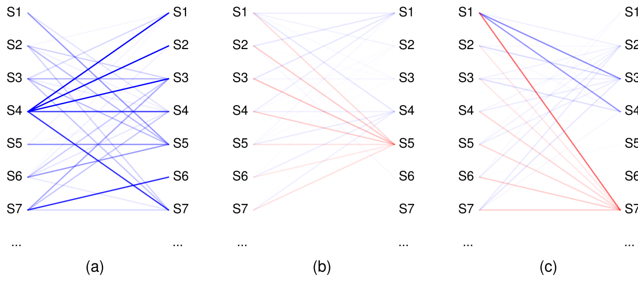


Fig. 6: NEURALPDA Attention Heads for the Example Java Program in Fig. 5: (a) L6-H4 (b) L4-H1 (c) L3-H3

such state-of-the-art, method-level DL-based vulnerability detection tool that takes as input a program semantics-capturing image extracted from the PDGs. In this experiment, we seek to determine how the PDGs predicted by NEURALPDA (say, PDG*) for *complete* methods affect the performance of downstream tasks. We will describe another vulnerability detection experiment for code snippets in Section IX.

We leverage VulCNN by taking as input both PDG* and the PDG derived from a program analysis tool (say, PDG$^\#$) for these methods, aiming to see how closely PDG* mimics PDG$^\#$ and approximates the performance of the vulnerability detection model. Mathematically, we formulate our task as follows:

$$0 < VD\{PDG^*\} \leq VD\{PDG^\#\} \quad (3)$$

where $VD\{.\}$ indicates the performance of the automated VD model. Here, if $VD\{PDG^*\} \lesssim VD\{PDG^\#\}$, we can establish the efficacy of the PDGs predicted by NEURALPDA for downstream SE tasks such as vulnerability detection.

A. Data Collection

We facilitate this study by using the VD dataset collected by Li *et al.* [37], which comprises of complete Java methods collected from eight large open-source Java projects. First, we filtered by projects, dedicating *avro*, *camel*, *hbase*, *hive*, *lucene-solr*, and *pig* for training purpose; *flink* and *cloudstack* for validation and testing respectively. Finally, we randomly selected an equal number of data samples from the vulnerable and benign method subsets in each of the splits, to obtain about 8K methods for training and about 1K each for testing and validation.

B. Experiment Setup

For all Java methods in the VD dataset, we extracted program dependencies (i.e., data and control-dependence edges) via Joern program analysis tool [19]. Next, using NEURALPDA trained on the Java dataset in Section VI-B (Table I), we generated PDGs (i.e., PDG*) for all the complete methods in the VD dataset. We then passed PDG$^\#$ and PDG* to VulCNN for vulnerability detection. VulCNN leverages centrality analysis to transform the PDGs into program semantics-capturing images. As a result, we generate two image datasets corresponding to PDG$^\#$ and PDG*, each of which are input to a convolutional neural network (CNN) for detecting the presence of vulnerabilities in complete code.

TABLE IX: Comparison of PDG$^\#$ (generated by PA tool) and PDG* (predicted by NEURALPDA) for method-level VD.

Methodology	TPR	TNR	F-Score
PDG$^\#$ + VulCNN	74.03	74.03	74.01
PDG* + VulCNN	73.27	73.27	73.26

C. Evaluation Metrics

We adopt the same metrics used by Wu *et al.* [25] to evaluate VulCNN, i.e., *true positive rate* (TPR) (also referred to as *Recall*), *true negative rate* (TNR), and *F-score*. Here, the positive label corresponds to the presence of a vulnerability in the method under study, while the negative label is given to a benign method. The better the performance of VulCNN, the closer PDG* mimics PDG$^\#$.

D. Experimental Results

Table IX shows VulCNN's performance in both settings, i.e., by using PDG$^\#$ and PDG*. We can observe that NEURALPDA predicts PDG* with an overall F-score of 91.13% (not shown). This further establishes the *generalizability of* NEURALPDA, more so, because the Java dataset that NEURALPDA was trained on, and the VD dataset comprising Java methods for which PDG*s were derived come from two *entirely distinct code corpora*. Moreover, VulCNN achieves an F-score of 73.26% using PDG*, which is a close approximate of the F-score of VulCNN using PDG$^\#$, 74.01%.

The PDGs predicted by NEURALPDA approximates the accuracy of those generated by program analysis for vulnerability detection on complete code by 98.98%.

IX. FRAGMENT-LEVEL VULNERABILITY DETECTION

Verdi *et al.* [1] manually inspected and reported 99 commonly used C/C++ code snippets from StackOverflow (S/O) answers as vulnerable. However, due to their incomplete nature, code snippets cannot automatically be analyzed for vulnerabilities. In this experiment, we design an ”in-the-wild” evaluation by: (a) first, leveraging NEURALPDA trained on complete C/C++ methods to predict PDGs for the incomplete C/C++ code snippets from S/O; (b) next, training VulCNN [25] to detect vulnerabilities in C/C++ code; (c) finally, making use of the predicted PDGs and trained VulCNN to check for vulnerabilities in the S/O code snippets.

A. Data Collection

In Section VI-A, we describe our expanded C/C++ dataset comprising of ∼50K methods, 26.3% of which are vulnerable. We split this dataset with a 80%-10%-10% ratio to train VulCNN. Of the 99 vulnerable code fragments collected by Verdi *et al.*, VulCNN fails to process 33 of them. We filter these out, and leverage NEURALPDA to predict PDGs for the remaining 66 code snippets.

B. Experiment Setup

First, for all C/C++ methods in the vulnerability detection dataset, we extract PDGs by leveraging the Joern program analysis tool [19]. We then train VulCNN on this dataset, achieving an F-score of 65.66% on the test set. Next, we utilize NEURALPDA that was trained on the C/C++ dataset in Section VI-E to predict the PDGs for the 66 code snippets from StackOverflow. Finally, we generate program semantics-capturing images corresponding to these predicted PDGs to input to the trained VulCNN model to discover the number of code snippets in which it can correctly identify vulnerabilities.

C. Experimental Results

We observed that VulCNN correctly identifies 14 out of the 66 code snippets as vulnerable. This is encouraging, more so because the C/C++ data that was used to train VulCNN (F-score=65.66%) does not cover all the vulnerabilities prevalent in the manually inspected S/O code snippets. The 52 mis-detected code snippets could be due to the inaccuracy of VulCNN. Besides, as in Section VIII, since the C/C++ dataset that was used to train NEURALPDA and the StackOverflow C/C++ code snippets do not possess any similarities, we can establish our tool's generalizability.

> PDGs predicted by NEURALPDA helps an automated VD tool discover 14 real-world vulnerable code fragments.

X. DISCUSSION

A. Time Complexity

For the complete methods in the Java dataset (see Section VI-A), on an average, Joern takes ~7.654s to generate the PDGs. In contrast, one can employ NEURALPDA to predict highly accurate PDGs in ~0.02s. To summarize, we observed a time markup of 380× by using NEURALPDA instead of Joern, which is especially useful for software engineering tasks that can tolerate low levels of imprecision.

B. Threats to Validity

Currently, we tested NEURALPDA on Java and C/C++. Performance could vary for other programming languages. We trained NEURALPDA by leveraging program dependencies extracted via a third-party program analysis (PA) tool. Thus, it can be impacted by the inaccuracies of the PA tool. Moreover, due to computational constraints, we limit the size of code snippets to 8 statements in both Java and C/C++ during training. For inference, the chunking strategy in Section IV-C can be used for longer code. However, even with training only on code with limited sizes, NEURALPDA can learn and generalize the dependencies with high accuracy as shown.

XI. RELATED WORK

In this paper, we seek inspiration for our problem setting from Chen and Manning [29], who first proposed a neural network-based approach to dependency parsing. The major benefits we envision to such a formulation include a significant speedup in dependency discovery and the extendibility of program dependence analysis to partial programs.

Specific to the SE domain, NEURALPDA is loosely related to works that leverage probabilistic models to enhance the program dependence graph (PDG). Probabilistic PDG [38] is an augmentation of the structural dependencies represented by a PDG with estimates of statistical dependencies between node states derived from test cases. Feng et al. [39] propose Error-Flow Graph as a Bayesian Network, constructed from the dynamic dependence graphs of the runs. Bayesian Network-based Program Dependence Graph (BNPDG) [40] is capable of inferring the dependencies across non-adjacent nodes. MOAD (Modeling Observation-based Approximate Dependency) [41] reformulates program dependency as the likelihood that one program element is dependent on another, instead of a boolean relationship. Lee [42] proposes a scalable approximate program dependence analysis by estimating the likelihood of dependence. It uses lexical analysis [43], partial observations on executions, and the merging of static and observation-based approaches. Those approaches leverage the knowledge from the executions to enhance the PDG for complete code. In contrast, we aim to use neural networks for deriving dependencies for both partial and complete code.

The recent success in machine learning has lead to strong interests in applying machine learning, especially deep learning, to programming language (PL) and software engineering (SE) tasks, such as automated correction for syntax errors [44], fuzz testing [45], program synthesis [46], code clones [47]–[49], program summarization [50], [51], code similarity [52], [53], probabilistic model for code [54], and path-based code representation, e.g., Code2Vec [53] and Code2Seq [55].

XII. CONCLUSION

This paper introduces a neural network-based approach for program dependence analysis of complete and partial code. We model this as a statement-pair dependence decoding problem, with the support of both intra-statement context learning and inter-statement context learning. NEURALPDA achieves high accuracy in generating CFG/PDGs for complete/partial code with much time efficiency. The F-score values for partial Java and C/C++ code range from 94.29%–97.17% and 92.46%–96.01%, respectively. We also show NEURALPDA's usefulness in vulnerability detection for partial code. The accuracy of the vulnerability detection tool utilizing the PDGs predicted by NEURALPDA is only 1.1% less than that utilizing the PDGs generated by a program analysis tool. Other SE applications that could tolerate some level of inaccuracies also benefit from NEURALPDA. Our work leads to a novel direction for improving program analysis (PA) for partial programs by combining neural networks with top-down PA techniques.

XIII. DATA AVAILABILITY

All code and data is available in our project website [36].

ACKNOWLEDGMENTS

This work was supported in part by the US National Science Foundation (NSF) grants CNS-2120386.

References

[1] M. Verdi, A. Sami, J. Akhondali, F. Khomh, G. Uddin, and A. K. Motlagh, "An empirical study of C++ vulnerabilities in crowd-sourced code examples," *IEEE Trans. Software Eng.*, vol. 48, no. 5, pp. 1497–1514, 2022. [Online]. Available: https://doi.org/10.1109/TSE.2020.3023664

[2] Flawfinder. [Online]. Available: http://www.dwheeler.com/FlawFinder

[3] Rats: Rough audit tool for security. [Online]. Available: https://code.google.com/archive/p/rough-auditing-tool-for-security/

[4] J. Viega, J.-T. Bloch, Y. Kohno, and G. McGraw, "Its4: A static vulnerability scanner for c and c++ code," in *Proceedings 16th Annual Computer Security Applications Conference (ACSAC'00)*. IEEE, 2000, pp. 257–267.

[5] Checkmarx. [Online]. Available: https://www.checkmarx.com/

[6] Hp fortify. [Online]. Available: https://www.hpfod.com/

[7] Coverity. [Online]. Available: https://scan.coverity.com/

[8] CWE-120: Buffer Overflow. [Online]. Available: https://cwe.mitre.org/data/definitions/120.html

[9] CWE-89: SQL Injection. [Online]. Available: https://cwe.mitre.org/data/definitions/89.html

[10] CWE-79: Cross-site Scripting. [Online]. Available: http://cwe.mitre.org/data/definitions/79.html

[11] CWE-290: Authentication Bypass by Spoofing. [Online]. Available: https://cwe.mitre.org/data/definitions/290.html

[12] Y. Li, S. Wang, and T. N. Nguyen, "Vulnerability detection with fine-grained interpretations," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2021, p. 292–303. [Online]. Available: https://doi.org/10.1145/3468264.3468597

[13] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet?" *IEEE Transactions on Software Engineering*, vol. 48, no. 09, pp. 3280–3296, sep 2022.

[14] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *Advances in Neural Information Processing Systems*, 2019, pp. 10 197–10 207.

[15] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "SySeVR: A framework for using deep learning to detect software vulnerabilities," *arXiv preprint arXiv:1807.06756*, 2018.

[16] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "VulDeePecker: A deep learning-based system for vulnerability detection," *arXiv preprint arXiv:1801.01681*, 2018.

[17] B. Dagenais and L. Hendren, "Enabling static analysis for partial java programs," *SIGPLAN Not.*, vol. 43, no. 10, p. 313–328, oct 2008. [Online]. Available: https://doi.org/10.1145/1449955.1449790

[18] H. Phan, H. A. Nguyen, N. M. Tran, L. H. Truong, A. T. Nguyen, and T. N. Nguyen, "Statistical Learning of API Fully Qualified Names in Code Snippets of Online Forums," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 632–642. [Online]. Available: https://doi.org/10.1145/3180155.3180230

[19] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *2014 IEEE Symposium on Security and Privacy*, 2014, pp. 590–604.

[20] M. Izadi, R. Gismondi, and G. Gousios, "Codefill: Multi-token code completion by jointly learning from structure and naming sequences," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 401–412. [Online]. Available: https://doi.org/10.1145/3510003.3510172

[21] S. Kim, J. Zhao, Y. Tian, and S. Chandra, "Code prediction by feeding trees to transformers," in *Proceedings of the 43rd International Conference on Software Engineering*, ser. ICSE '21. IEEE Press, 2021, p. 150–162. [Online]. Available: https://doi.org/10.1109/ICSE43902.2021.00026

[22] O. Chaparro, C. Bernal-Cárdenas, J. Lu, K. Moran, A. Marcus, M. Di Penta, D. Poshyvanyk, and V. Ng, "Assessing the quality of the steps to reproduce in bug reports," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 86–96. [Online]. Available: https://doi.org/10.1145/3338906.3338947

[23] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Bug localization with combination of deep learning and information retrieval," in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, 2017, pp. 218–229.

[24] A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "A large-scale study on repetitiveness, containment, and composability of routines in open-source projects," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 362–373. [Online]. Available: https://doi.org/10.1145/2901739.2901759

[25] Y. Wu, D. Zou, S. Dou, W. Yang, D. Xu, and H. Jin, "VulCNN: An image-inspired scalable vulnerability detection system," 2022.

[26] J. Fan, Y. Li, S. Wang, and T. Nguyen, "A C/C++ code vulnerability dataset with code changes and cve summaries," in *The 2020 International Conference on Mining Software Repositories (MSR)*. IEEE, 2020.

[27] [Online]. Available: https://stackoverflow.com/a/10702464

[28] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, p. 319–349, jul 1987. [Online]. Available: https://doi.org/10.1145/24039.24041

[29] D. Chen and C. Manning, "A fast and accurate dependency parser using neural networks," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 740–750. [Online]. Available: https://aclanthology.org/D14-1082

[30] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," *CoRR*, vol. abs/1706.03762, 2017. [Online]. Available: http://arxiv.org/abs/1706.03762

[31] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[32] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized BERT pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.

[33] M. Allamanis and C. Sutton, "Mining Source Code Repositories at Massive Scale using Language Modeling," in *The 10th Working Conference on Mining Software Repositories*. IEEE, 2013, pp. 207–216.

[34] T. Mikolov, K. Chen, G. S. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," 2013. [Online]. Available: http://arxiv.org/abs/1301.3781

[35] K. Clark, U. Khandelwal, O. Levy, and C. D. Manning, "What does BERT look at? an analysis of BERT's attention," in *Proceedings of the 2019 ACL Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*. Florence, Italy: Association for Computational Linguistics, Aug. 2019, pp. 276–286. [Online]. Available: https://aclanthology.org/W19-4828

[36] [Online]. Available: https://github.com/deeppda-icse23/DeepPDA/

[37] Y. Li, S. Wang, T. N. Nguyen, and S. Van Nguyen, "Improving bug detection via context-based code representation learning and attention-based neural networks," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3360588

[38] G. K. Baah, A. Podgurski, and M. J. Harrold, "The probabilistic program dependence graph and its application to fault diagnosis," in *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ser. ISSTA '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 189–200. [Online]. Available: https://doi.org/10.1145/1390630.1390654

[39] M. Feng and R. Gupta, "Learning universal probabilistic models for fault localization," in *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, ser. PASTE '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 81–88. [Online]. Available: https://doi.org/10.1145/1806672.1806688

[40] X. Yu, J. Liu, Z. Yang, and X. Liu, "The Bayesian Network Based Program Dependence Graph and Its Application to Fault Localization," *J. Syst. Softw.*, vol. 134, no. C, p. 44–53, dec 2017. [Online]. Available: https://doi.org/10.1016/j.jss.2017.08.025

[41] S. Lee, D. Binkley, R. Feldt, N. Gold, and S. Yoo, "MOAD: Modeling observation-based approximate dependency," in *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2019, pp. 12–22.

[42] S. Lee, "Scalable and approximate program dependence analysis," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 162–165. [Online]. Available: https://doi.org/10.1145/3377812.3381392

[43] S. Lee, D. Binkley, N. Gold, S. Islam, J. Krinke, and S. Yoo, "Evaluating lexical approximation of program dependence," *J. Syst. Softw.*, vol. 160, no. C, feb 2020. [Online]. Available: https://doi.org/10.1016/j.jss.2019.110459

[44] S. Bhatia and R. Singh, "Automated correction for syntax errors in programming assignments using recurrent neural networks," *CoRR*, vol. abs/1603.06129, 2016. [Online]. Available: http://arxiv.org/abs/1603.06129

[45] J. Patra and M. Pradel, "Learning to fuzz: Application-independent fuzz testing with probabilistic, generative models of input data," 2016.

[46] M. Amodio, S. Chaudhuri, and T. W. Reps, "Neural attribute machines for program generation," *CoRR*, vol. abs/1705.09231, 2017. [Online]. Available: http://arxiv.org/abs/1705.09231

[47] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE. New York, NY, USA: Association for Computing Machinery, 2016, p. 87–98. [Online]. Available: https://doi.org/10.1145/2970276.2970326

[48] R. Smith and S. Horwitz, "Detecting and measuring similarity in code clones," 2009.

[49] L. Li, H. Feng, W. Zhuang, N. Meng, and B. Ryder, "CCLearner: A Deep Learning-Based Clone Detection Approach," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2017, pp. 249–260.

[50] M. Allamanis, H. Peng, and C. A. Sutton, "A convolutional attention network for extreme summarization of source code," *CoRR*, vol. abs/1602.03001, 2016. [Online]. Available: http://arxiv.org/abs/1602.03001

[51] L. Mou, G. Li, Z. Jin, L. Zhang, and T. Wang, "TBCNN: A tree-based convolutional neural network for programming language processing," *CoRR*, vol. abs/1409.5718, 2014. [Online]. Available: http://arxiv.org/abs/1409.5718

[52] G. Zhao and J. Huang, "Deepsim: Deep learning code functional similarity," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: ACM, 2018, pp. 141–151. [Online]. Available: http://doi.acm.org/10.1145/3236024.3236068

[53] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *CoRR*, vol. abs/1803.09473, 2018. [Online]. Available: http://arxiv.org/abs/1803.09473

[54] P. Bielik, V. Raychev, and M. Vechev, "Phog: Probabilistic model for code," in *Proceedings of The 33rd International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, M. F. Balcan and K. Q. Weinberger, Eds., vol. 48. New York, New York, USA: PMLR, 20–22 Jun 2016, pp. 2933–2942. [Online]. Available: http://proceedings.mlr.press/v48/bielik16.html

[55] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," *arXiv preprint arXiv:1808.01400*, 2018.