IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 13 November 2023, accepted 20 November 2023, date of publication 1 December 2023, date of current version 6 December 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3338162

RESEARCH ARTICLE

Software Vulnerability Detection Using Informed Code Graph Pruning

JOSEPH GEAR[1], YUE XU[1], (Member, IEEE), ERNEST FOO[2], (Member, IEEE), PRAVEEN GAURAVARAM[3], ZAHRA JADIDI[2], (Member, IEEE), AND LEONIE SIMPSON[1]
[1]School of Computer Science, Queensland University of Technology, Brisbane, QLD 4000, Australia
[2]School of Information and Communication Technology, Griffith University, Brisbane, QLD 4111, Australia
[3]Cyber Security Research and Innovation, Tata Consultancy Services Ltd. (TCS), Brisbane, QLD 4000, Australia

Corresponding author: Joseph Gear (n9951610@qut.edu.au)

This work was supported by the Cyber Security Research Centre Ltd. funded by the Australian Government's Cooperative Research Centers Program.

ABSTRACT Security vulnerabilities in source code are a growing problem which can lead to financial, reputation, physical, and even human damage when exploited by malicious actors. The use of deep learning to locate these vulnerabilities before they are deployed is currently a major area of security research. Code graphs such as Abstract Syntax Trees and Code Property Graphs are commonly used as input data to deep learning models as they are highly expressive of the code's function. However, as the length of source code increases, so too does the computational cost of vulnerability detection models. Typically this cost is managed by using only a sample of the input graph, however this is often done randomly and with little consideration for the lost information. Little work has been done to explore informed heuristic-based pruning methods that can be used to reduce graph size to manageable levels by removing information irrelevant to vulnerabilities, while preserving relevant information. We present "Semantic-enhanced Code Embedding for Vulnerability Detection" (SCEVD), a deep learning model for vulnerability detection that seeks to fill these gaps by using more detailed information about code semantics to select vulnerability-relevant features from code graphs. We propose several heuristic-based pruning methods, implement them as part of SCEVD, and conduct experiments to verify their effectiveness. Our heuristic-based pruning improves on vulnerability detection results by up to 12% over the baseline pruning method.

INDEX TERMS Code representation, deep learning, source code semantics, vulnerability detection.

I. INTRODUCTION

Security vulnerabilities in source code are growing in both pervasiveness and impact. As an increasing number of systems and components of society become digitised and networked, the incentives for malicious actors to exploit weaknesses and flaws in these systems increase as well. Vulnerability Detection (VD) is the task of identifying these vulnerabilities before they can be exploited so that they can be corrected. Machine learning (ML) and deep learning (DL) have been growing in popularity as methods for solving this problem [6], [24], [51]; however many open questions still exist. In order for source code to be interpretable by a DL model, it must first be represented numerically in a process known as *code embedding*. Producing more expressive code embeddings that capture the underlying patterns in code better can reduce the need for complex VD models and improve detection accuracy [24]. This process is critical for ML-based VD, however it is also a challenging one due to the complexities of source code and the difficulties in defining and expressing its "meaning". This remains an open problem and one that must be addressed if ML models are to be deployed for the purposes of VD.

The highly functional and structured nature of code lends itself to a graphical representation using *code graphs*, usually either Abstract Syntax Trees (ASTs) or Code Property Graphs (CPGs), rather than a simple sequence of tokens (which are to source code as words are to sentences in

The associate editor coordinating the review of this manuscript and approving it for publication was Porfirio Tramontana.

natural language. E.g., "int", "x", "=", or "5") akin to popular methods used in Natural Language Processing (NLP). Although token-based code embeddings have been used for code VD in the past [13], [16], [20], [21], [56], [62], graphical representations have shown increasing promise in recent years, and their structure offers distinct advantages over token-based representations of code [4], [7], [10]. For token-based embedding methods, the features used for learning are the tokens themselves (as well as combinations of tokens such as code statements), whereas for graph-based embedding methods, the features can be more varied, including (but not limited to) the tokens represented by graph nodes, edges connecting neighbouring nodes in the graph, and paths between distant nodes.

Lin et al. [24] posit that token-based embeddings have difficulty capturing the meaning of tokens that exists independent of the tokens' context, and that graph-based embedding methods may be better at expressing this information in source code. Because of this, code graph learning has seen continued use and exploration in the existing literature [2], [4], [7], [10], [16], [56]. However, owing to the increased complexity of graphs as a data type, it is necessary to choose the features from these graphical representations that best represent the underlying patterns in code which lead to security vulnerabilities [28], [45], [49].

We believe that this feature selection problem has received insufficient attention from previous studies in the domain of source code VD. For a given *code snippet*, that is, a sequence of code statements that work together to produce a result, it is necessary to determine which features of that snippet are most indicative of the presence or absence of a vulnerability. For many code snippets, a large amount of code that does not contain a vulnerability may be present (e.g., code statements that do not interact with buffers cannot cause buffer overflows on their own), which may confuse ML models or obfuscate actual vulnerabilities [10]. In fact, some studies attempt to provide VD at a line-level granularity (as opposed to function- or file-level), as they consider vulnerabilities to be small enough relative to a whole function to be isolated to individual lines [13], [16].

```
1  char * readSocket(int socket) {
2      char * buff;
3      buff = (char *) malloc(100);
4      int i = 0;
5      while (isAvailable(socket) || i < 100) {
6          char c = readSerial(socket);
7          buff[i] = c;
8          i++;
9      }
10     return buff;
11 }
```

LISTING 1: A vulnerable readSocket function.

Consider the example C/C++ function `readSocket` in Listing 1. This function is an example of Common Weakness Enumeration 119 (CWE-119), "Improper Restriction of Operations within the Bounds of a Memory Buffer", more commonly known as a buffer overflow. The task of this function is to read the characters sent over a given serial connection and to store them in a buffer. This `readSocket` function possesses a buffer overflow vulnerability because it incorrectly checks that the input it has received is sufficiently small to fit in the memory it has reserved. In this case, replacing the 'or' operator (||) on line 5 with an 'and' operator (`&&`) would correct the vulnerability. In this example, only one token (and consequently, only one node in the graph) need be changed for the vulnerability to be removed. This example is extreme, as only a single node is responsible for the vulnerability, and many functions may have a greater number of relevant nodes; however this proportion of relevant nodes to irrelevant ones is still likely to be small for most code snippets.

Most features of the code graph that may be produced from this snippet are irrelevant to the vulnerability of this code. Learning from every possible feature in a code graph is prohibitively computationally expensive for most non-trivial code snippets. Therefore, code graphs must be truncated to a fixed size. However, existing code embedding models, such as *code2vec* [2] and *VulSniper* [10], do not give particular attention to *which* features should be considered representative for the purpose of VD. Without a method for assessing the vulnerability relevancy of features in a code graph when generating the code representation, it would be possible to omit some important features (e.g., tokens and paths) that cause vulnerability. Omitting these features would lead the representation provided to an ML-based VD model to lack the information necessary to determine whether the code is vulnerable, yielding inaccurate results [51].

In this paper, we present Semantic-enhanced Code Embedding for Vulnerability Detection (SCEVD), a DL-based model for detecting code vulnerabilities that addresses the lack of attention given to selecting the representative features of code snippets for VD. SCEVD highlights the properties of nodes in code graphs and the relationships between nodes indicating that a given node is more or less relevant to the representation of the overall snippet. SCEVD detects code vulnerabilities by prioritising the most relevant nodes. We refer to our model as *semantic-enhanced* because we use semantic information (i.e., information regarding the underlying meaning of code) from code graphs, both their nodes and edges, to inform the prioritisation of graph features.

Our contributions in this paper are as follows:

- We propose a set of heuristics to select the most representative features in code graphs. These heuristics are designed to focus on vulnerability-relevant features. We test these heuristics on several software vulnerability datasets and find that they improve SCEVD's ability to detect vulnerabilities compared with random pruning.
- We apply SCEVD to both ASTs and CPGs and evaluate how effectively our model is able to detect vulnerabilities in each code graph type.

- We examine the code graph features to which SCEVD assigns the most attention when detecting vulnerabilities and find that these features align with those highlighted in a human evaluation of the code.
- We identify properties of each dataset which correlate with the strengths of particular pruning heuristics.
- We compare SCEVD to other models and tools for detecting vulnerabilities and their graph pruning methods. We find that SCEVD tends to outperform these models, and that the application of pruning heuristics to these models also improves their performance.

The remainder of this paper is structured as follows: **Section II** provides preliminaries and background regarding the problem of embedding graphical representations of source code, what those representations are, and how existing source code embedders use graphs to extract the features mentioned above. We also provide a motivating example of SCEVD's usage. **Section III** describes SCEVD, **Section IV** describes our experimental methodology, **Section V** shows our experimental results and our findings, and **Section VI** covers related work. **Section VII** discusses the limitations of our study and **Section VIII** concludes this paper.

II. BACKGROUND AND MOTIVATION

In this section, we establish the key concepts behind graphical code representation and discuss existing studies that utilise these representations for code embedding.

A. GRAPHICAL CODE REPRESENTATION

Graphically representing source code has a number of advantages compared to the use of code tokens. NLP techniques appear applicable to source code because of the similarities between code and natural language (that is, human-readability), but there are key differences between the two domains. Notably, the structure of source code is more rigid than natural language and the meaning of individual tokens can be more specific than the meaning of words in language. For example, keywords like `return` or `else` in C (and other programming languages) have absolute meanings independent of context and both strongly affect the meaning of a code snippet. This structured nature of code has led many to focus on the use of graphs in code representation. Yang et al. [48] find at least 59 studies which utilise code graphs as input for ML models.

The tasks to which code graph learning is applied vary widely. These tasks include: function and variable name prediction [1], [2], [26], comment generation [40], [60], and code clone detection [38], [52], [55], etc. Similarly, code graph learning has been used for code vulnerability detection [4], [7], [10], [43].

B. SYNTAX AND SEMANTICS

A common distinction made when studying ML for source code (as well as NLP) is between the *syntax* of a code snippet and the *semantics* of a code snippet. We consider semantic information to describe the underlying meaning of the code. In the case of source code, the 'meaning' is what is done by the computer when the code is executed. The semantic meaning of a code snippet is generally abstract. For example, the 'meaning' of lines 2-4 of the code in Listing 2 is that the computer will reserve space in memory for three variables: two arrays of characters and an integer. These variables are `source` (which reserves 9 bytes of memory and fills them with the string "`too long`"), `dest` (which reserves 5 bytes but does not fill them), and `trigger`, which has an initial value of 5.

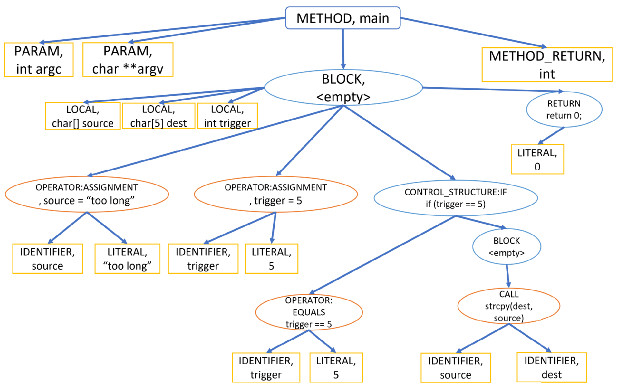

FIGURE 1. Abstract syntax tree for the code snippet in Listing2.

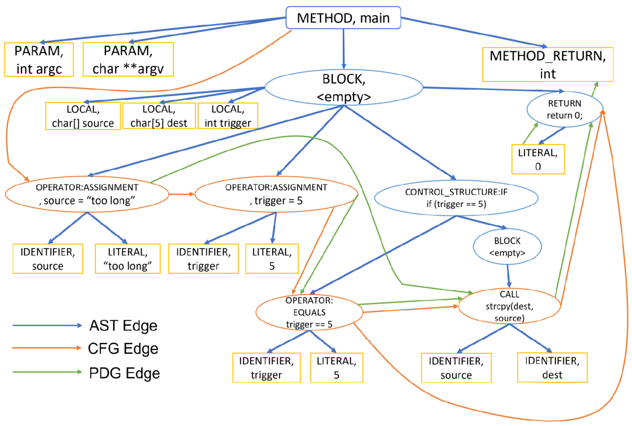

FIGURE 2. Code property graph for the code snippet in Listing 2.

Syntactic information refers to how the code is structured at a fine-grained level. Syntactically, there is no direct relationship between the variable definitions in lines 2 and 4; however there is a semantic relationship, as once the statement in line 2 is executed, control is passed to the statement in line 4. In short, syntactic information is what can be *seen* when the code is read, whereas semantic information is what is *done* when the code is executed.

C. ABSTRACT SYNTAX TREES

Abstract Syntax Trees (ASTs) explicitly represent the syntactic relationships between elements of code. Each node in the tree denotes a construct in the source code. Figure 1

```
1 int main(int argc, char **argv) {
2     char source[] = "too long";
3     char dest[5];
4     int trigger = 5;
5     if (trigger == 5) {
6         strcpy(dest, source);
7     }
8     return 0;
9 }
```

LISTING 2: A vulnerable function.

shows the AST for the example function in Listing 2. The `IF` node has two children: a `COMPARISON` node representing the condition to be checked, and a `BLOCK` node containing the instructions to be executed when that condition is met. An AST does not make the semantic connections between statements in source code explicit; however, the AST nodes do contain semantic information as they detail variable types, the role of the node within the code, etc. [46]. ASTs can also possess many terminal nodes, sometimes called the 'leaves' of the tree, which are represented by yellow boxes in Figure 1. These terminal nodes represent individual tokens in the code, whereas the intermediate nodes represent combinations of these tokens, such as statements or lines of code.

Definition 1 (AST): An AST is a tuple $\langle N, V\rangle$ where N is a set of nodes and $V \subset N \times N$ is a set of 2-tuples. $\forall(a, b) \in V$, $a, b \in N$, (a, b) is a pair of directly connected nodes, a is a parent node, and b is a child node.

D. CODE PROPERTY GRAPHS

Code Property Graphs (CPGs), proposed by Yamaguchi [46], are graphical representations of code that combine the AST, Control Flow Graph (CFG), and Program Dependency Graph (PDG). Additional edges for the CFG and PDG are added between the AST nodes to represent how control is passed through the program (CFG edges) and how information is passed and manipulated through the program (PDG edges). This additional information can be used to show direct relationships between AST nodes, which would be clear to a human when reading the code, but may be far apart from one another in an AST. For example, in Figure 1, no explicit relationship is drawn between the assignment `int trigger = 5;` statement in line 4 and the comparison `trigger == 5` in line 5 of Listing 2. However, the CPG for Listing 2 illustrated in Figure 2 draws a CFG edge directly between these two statements, showing that their relationship is stronger than that which an AST alone would describe. We henceforth refer to AST edges as describing syntactic relationships between nodes and the other CPG edges (CFG, PDG) as describing semantic relationships. Because the CPG uses the AST as a base, it also possesses terminal nodes without children; however, some of the terminal AST nodes are assigned outgoing edges to other nodes in the CPG. While these nodes still represent only individual code tokens (and are still represented by yellow boxes in Figure 2), they are no longer necessarily 'terminal' (as they now have outgoing edges), so we will henceforth refer to these nodes as '*token*' nodes. For example, in Figure 1, the '`LITERAL: 0`' node whose parent is the `RETURN` node is a 'terminal' node, as it has no children. In Figure 2, this node is given an PDG edge as part of the CPG and is therefore no longer a terminal node, so we refer to it as a 'token' node.

Definition 2 (CPG): a CPG is a tuple $\langle N, C\rangle$ where N is a set of nodes and $C \subset N \times N \times K$ is a set of 3-tuples. $\forall(x_s, x_e, k) \in C$, $x_s, x_e \in N$, (x_s, x_e) is a pair of adjacent nodes with a directed edge from x_s to x_e, and $k \in K$ is an edge type. For a standard CPG, $K = \{AST, CFG, PDG\}$.

E. CODE2VEC AND PATH-CONTEXTS

code2vec is a DL model designed to embed ASTs as vectors. Alon et al. [2] proposed code2vec to embed ASTs for function name generation; however, it is intended to be a generalisable embedder that can be used for a variety of downstream tasks (e.g., automated function naming, code duplication detection, and code authorship prediction).

code2vec takes the AST of a code snippet and extracts representations of every pair of terminal nodes and the path between them through the tree. Each path is a sequence of nodes and edges that connects the two terminal nodes.

Each code snippet is represented as a bag of these "path-contexts". A path-context is defined as "a triplet $\langle x_s, p, x_t\rangle$" [2] where x_s and x_t are the tokens held by the terminal nodes $s \in N$ and $t \in N$ at the start and the end of the path p through the AST. For example, the path-context for the terminal node `IDENTIFIER: trigger` and the terminal node `LITERAL: 5` on line 4 of Listing 2 would be $\langle \texttt{trigger}, (Identifier \uparrow AssignmentOperator \downarrow Literal), 5\rangle$.

For each path-context, a vector representation $c \in \mathbb{R}^{3d}$ is produced by concatenating the vector representations of x_s, p, and x_t where d is the length of the vector representations of x_s, p, and x_t. The vector representations for each of the constituent parts of the path-contexts are drawn from two vocabulary matrices: $node\_vocab \in \mathbb{R}^{|X|\times d}$ and $path\_vocab \in \mathbb{R}^{|Y|\times d}$, where X and Y are a set of representations of nodes and a set of representations of unique paths in the dataset, respectively. These vocabulary matrices are initialised randomly and learned during training. For a given path-context, the vectors $node\_vocab_s$ and $node\_vocab_t$ can be thought of as answers to the question "how can the nodes s and t be represented?" and the vector $path\_vocab_p$ is the answer to the question "how can the relationship between s and t be represented?"

Once path-context representations have been produced for an entire snippet, they are combined into a single vector representation $v_i \in \mathbb{R}^{3d}$ for snippet $f_i \in D$ in dataset D using an attention mechanism, which acts as a weighted sum in which the weights for each path-context are learned during training. In this way, the attention mechanism learns which specific path-contexts are most relevant to the representation of the entire snippet and which are not.

Although this method is effective for automatically learning representations for individual terminal nodes and path-contexts in code graphs, as well as in aggregating the path-context representations using an attention mechanism to produce the overall snippet representation, it is not computationally feasible to learn representations and attention weights for *every* path-context in every snippet in the dataset. The number of path-contexts for a snippet grows rapidly with the number of terminal nodes in the snippet code graph. The number of path-contexts will be equal to the sum of all numbers from 1 to the number of terminal nodes in the graph ($pc = \sum_{i=1}^{tn} i$, where pc is the number of path-contexts in the graph and tn is the number of terminal nodes in the graph). Thus, for very large and complex snippets, hundreds of terminal nodes and thousands or tens of thousands of path-contexts can exist, however even for relatively short snippets, this can be a problem. A code graph with only 15 terminal nodes (as Listing 2 does) will have 105 path-contexts, one with 21 will have 210, and after just 46 terminal nodes the number of path-contexts will exceed 1000.

Because the path-attention model used by code2vec takes a fixed number of path-contexts for training, this means that every graph in the training dataset (and every graph used for inference once training is complete) would need to be padded to the size of the largest graph in the dataset. While this would provide the attention module with all the information available to produce graph vector v_i, this presents two major problems. The first is that, such a model, once trained, would be incapable of making predictions about graphs larger than its largest training graph, limiting its flexibility. The second is that graphs which are outliers in terms of size can be extremely large. For the datasets from which Listings 2 and 3 are drawn,[1] the largest graph has more than 21,000 path-contexts. This means that if $d = 128$ (as in [2]) the combined-context embeddings for a single graph in the dataset would take up more than 10 megabytes of memory.[2] For large datasets or those with very large code snippets, this memory impact is prohibitively high and makes model training impossible.

To deal with the scalability issue, code2vec selects a subset of path-contexts of size m ($m = 200$ in [2]) at random to represent each snippet before training commences (padding snippets with fewer than 200 path-contexts); however this process does not guarantee that the path-contexts selected are actually representative of the snippet in a way that is meaningful for many downstream tasks, as it does not consider node or edge properties. For the VD, the presence of a vulnerability may be indicated by only a very small number of critical path-contexts, which may be overlooked if the path-contexts are sampled randomly. Thus, the value of m is a trade-off between computational resources and model accuracy. Where m is higher, the resources and time needed to train code2vec increase, however so does the accuracy.

While the principles behind bags-of-path-contexts as representations for code snippets are promising for the purposes of code embedding for VD, random path-context selection and reliance on only the structural information of ASTs as code graphs (rather than the more expressive CPGs) are weaknesses that we believe can be overcome to produce code embeddings that are more applicable to VD if the properties of nodes and edges are considered.

```
void bad()
{
    int * data;
    data = NULL;
    data = (int *)ALLOCA(10);
    {
        int source[10] = {0};
        size_t i;
        for (i = 0; i < 10; i++)
        {
            data[i] = source[i];
        }
    }
}
```

LISTING 3: A vulnerable function.

```
void good()
{
    int * data;
    data = NULL;
    data = (int *)ALLOCA(10*sizeof(int));
    {
        int source[10] = {0};
        size_t i;
        for (i = 0; i < 10; i++)
        {
            data[i] = source[i];
        }
    }
}
```

LISTING 4: A safe function.

F. MOTIVATING EXAMPLE

To demonstrate the use and value of SCEVD, we present a motivating example of the vulnerability detection process using SCEVD. Consider the code in Listings 3 and 4 (taken from the SARD Juliet dataset[3]). In both functions, memory on the stack is allocated to the buffer `data`. In Listing 3 this is ten bytes of memory, while in Listing 4, it is ten integers' worth of memory (as an integer in C takes up four bytes, which is 40 bytes of memory). Afterwards, each function operates identically: declaring a second buffer `source` that is filled with ten `0` integers, using a `for` loop to loop through the first ten indices of each buffer, and assigning the contents of `source` at each index to `data` at the same index. Because `data` and `source` are both integer buffers, accessing index `i` means accessing the $4i$-th byte of the buffer. Thus, when

[1]The Software Assurance Reference Dataset (SARD) Juliet, specifically the CWE-121 dataset, which is a subset of CWE-119.

[2]Where each combined-context vector is a 128-long 32-bit floating point vector, the memory impact for a graph is $4\text{ bytes per float} \times 128\text{ floats per path-context} \times 21,000\text{ path-contexts} = 10,752,000\text{ bytes} = 10.752\text{ Mb}$.

[3]CWE121_Stack_Based_Buffer_Overflow__CWE131_loop_01.c.

the operation `data[1] = source[1]` is performed, the values at bytes 4-7 of `source` are read and copied into bytes 4-7 of `data`.

For Listing 3, this results in a Stack-Based Buffer Overflow (CWE-121) when $i > 1$, because when $i = 2$ bytes 8-11 of `source` will be copied into the buffer at `data`, which only has ten bytes allocated to it, meaning that the two bytes immediately after `data`'s allocation will be overwritten, as will the following 28 as the loop continues and i grows. If any of the overwritten bytes contain critical information, or if the contents of `source` can be tailored to contain malicious instructions, this would result in a security vulnerability. The same is not true for Listing 4, which allocates sufficient space on the stack for the entire contents of `source` to be copied into `data` without any memory overwrites.

To detect this vulnerability, a reader would need to identify that the maximum value of the loop counter `i` results in bytes outside the safe allocation of `data` being written to. We posit that SCEVD is capable of drawing this connection, and will analyse its results on these example functions in **Section V**.

III. SEMANTIC-ENHANCED CODE EMBEDDINGS FOR VULNERABILITY DETECTION

A. NODE-PROPERTIES AND RELATIONSHIP-PROPERTIES

We consider two types of information to be found in code graphs: *node-property* information and *relationship-property* information. Node-properties refer to the information contained in each node of the graph and for SCEVD we consider two key pieces of node-property information: the node's type, and the node's code value. Thus, we consider two node-property functions: *node_type* and *node_value*. For each node n, $node\_type(n)$ and $node\_value(n)$ return n's type and value, respectively. For example, for the root node of the graph in Figure 2, $node\_type(n) = METHOD$ and $node\_value(n) =$ `main`. While the *node_value* of a node is based on the code it represents and can therefore vary widely from node to node, there is a limited set of possible CPG node types. The CPG specification we use [47] defines several node types, including `BLOCK`, `CONTROL_STRUCTURE`, `CALL`, and `LITERAL`.

Relationship-properties refer to information regarding the relationships between nodes in the graph. Note that these relationships can refer to more distant relationships than simply the edge between two adjacent nodes, but can also refer to the relationship between *any* pair of nodes in the graph, usually described as the path through the graph between those nodes. In this paper, we consider four node-relationship-properties between the start and end nodes of a path: $path\_length(p)$, $cfg\_edge\_count(p)$, $ast\_edge\_count(p)$, and $pdg\_edge\_count(p)$, representing the number of edges, number of CFG edges, number of AST edges, and number of PDG edges between the start and end nodes of a path p, respectively.

Both types of information contain important semantic information, but existing works have focused largely on learning from the structure of the graph. While this gives them the ability to learn which structural elements of code snippets lead to vulnerabilities, node-property information and detailed information about node-relationship-properties are often overlooked.

With SCEVD (depicted at a high level in Figure 3), we seek to take advantage of both types of code graph information to address the weaknesses of previous works. Specifically, we use the node-property and node-relationship information provided by CPGs to produce more semantically-informed representations of code snippets, and we use some common knowledge of software coding and security to select code graph features (in our case, path-contexts) that are more relevant to the presence or absence of vulnerabilities in a code snippet.

B. BAG-OF-PATH-CONTEXTS

First, we produce a bag-of-path-contexts for each code snippet. code2vec uses ASTs as input; thus, there is only one possible path between each pair of terminal nodes. However we also use CPGs, and as such multiple paths may exist between any pair of nodes. As discussed above, this process amounts to producing node-relationship information, the result answering the question "how can we best describe the relationships between nodes in this graph?"

Therefore, we use Dijkstra's algorithm to find the shortest path between pairs of token nodes in the CPG (the "Pathfinder" module in Figure 3. The length of each path is counted as the number of CPG edges in the path. We choose to use the shortest path between each pair of token nodes, as the purpose of the CPG is to bring semantically-related nodes into close proximity to one another; therefore the clearest semantic relationship between two token nodes can be described as the shortest path between them in the CPG. Because the semantic edges drawn to produce the CPG typically create "shortcuts" between nodes, the shortest path between two nodes will usually follow these semantic edges where possible, rather than the syntactic edges of the AST, even if no artificial priority is given to semantic edges.

C. HEURISTIC-BASED PATH PRUNING

While SCEVD's DL-module uses an attention mechanism (see *Embedding* for more details), which learns which path-contexts to prioritise when producing combined code embeddings, it is computationally infeasible to allow the model to use *every* possible path-context as input for learning. The number of path-contexts produced to represent a code snippet grows exponentially with the number of token nodes in that snippet, so large snippets can contain many thousands of path-contexts.

Consequently, it is necessary to select a subset of path-contexts that represent the snippet. Rather than randomly sampling a set of path-contexts, as in [2], we choose

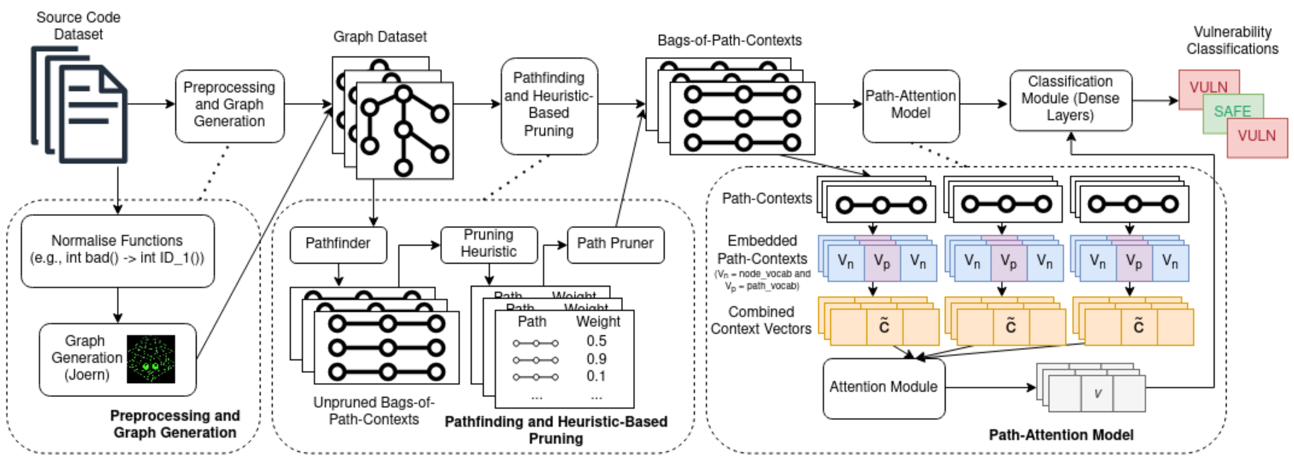
Source Code Dataset
Preprocessing and Graph Generation
Graph Dataset
Pathfinding and Heuristic-Based Pruning
Bags-of-Path-Contexts
Path-Attention Model
Classification Module (Dense Layers)
Vulnerability Classifications
VULN
SAFE
VULN
Normalise Functions (e.g., int bad() -> int ID_1())
Graph Generation (Joern)
Preprocessing and Graph Generation
Pathfinder
Pruning Heuristic
Path Pruner
Path
Weight
0.5
0.9
0.1
Unpruned Bags-of-Path-Contexts
Pathfinding and Heuristic-Based Pruning
Path-Contexts
Embedded Path-Contexts
(Vn = node_vocab and Vp = path_vocab)
Combined Context Vectors
Attention Module
Path-Attention Model

FIGURE 3. SCEVD: Semantic-enhanced Code Embedding for Vulnerability Detection. This diagram shows our entire proposed process for vulnerability detection. Preprocessing and Graph Generation is described in Section IVA, Pathfinding and Heuristic-Based Pruning is described in Sections III-B and III-C, and the Path-Attention Model is described in Section III-D.

to use a heuristic-based approach. Although Alon et al. were able to achieve good results using random sampling, they did so for the purpose of function name prediction. This is a reasonable approach for name prediction because functions are ideally named according to their *intent*. The programming patterns used to create the `readSocket` function in Listing 1 are likely to be distinct enough from functions with other purposes that only a fraction of these patterns need to be used as input for code2vec to determine that the function is supposed to read from a network socket. However, as mentioned, the buffer overflow vulnerability in `readSocket` exists only because of a single node as represented in the CPG, and it should be named `readSocket`, regardless of whether it contains a security vulnerability. For code snippets that require truncation, these critical vulnerabilities may be overlooked if a small set of path-contexts are randomly sampled, and it is therefore necessary to guide this selection process towards candidate path-contexts considered more likely to be relevant to the presence or absence of vulnerabilities.

The selection of which path-contexts to use to represent a snippet amounts to selecting which nodes (and consequently, which code features) should be considered the most important for VD. If a path-context is the answer to a question about the relationship between two nodes, the path-contexts selected to represent the snippet answer the question "which node pairs are most important to consider in order to describe this snippet?" Henceforth, we refer to this selection process as *path pruning* and the heuristic used to select representative path-contexts as the *pruning heuristic*.

Let P be a set of all the path-contexts generated by the pathfinding algorithm. A pruning heuristic $\epsilon : P \mapsto \mathbb{R}$ is a function that takes a single path-context $p \in P$ and assigns a weight $w \in \mathbb{R}$. In this paper, we propose four pruning heuristics to determine how best to guide the selection process towards path-contexts that are most relevant to the presence or absence of vulnerabilities in a code snippet. The proposed pruning heuristics are defined below, where $p \in P$ is a path-context.

Random weight heuristic: Assigning a random value to p. This is the baseline method used by code2vec.

$$\epsilon_{random}(p) = \text{rand}(p)$$

CPG-ratio heuristic: Selecting candidate path-contexts based on the ratio of semantic edges (CFG, PDG, etc.) to syntactic edges (AST), assuming that nodes that are predominantly related by semantic information are more likely to be relevant to the overall representation of the function.

$$\epsilon_{ratio}(p) = \frac{cfg\_edge\_count(p) + pdg\_edge\_count(p)}{ast\_edge\_count(p) + 1}$$

Inter-node heuristic: Selecting candidate path-contexts based on the properties of the nodes along the path, assuming that the presence of certain properties in nodes along the path is more likely to be relevant to the overall representation of the function.

$$\epsilon_{interNode}(p) = \frac{\sum_{i=1}^{path\_length(p)} \phi(n_i)}{path\_length(p)}$$

where $\phi(n)$ is the inter-node value for node n, which is defined as:

$$\phi(n) = \begin{cases} 1 & \text{if } node\_type(n) \in target\_types \\ 0 & else \end{cases} \tag{1}$$

where $target\_types = \{\texttt{CALL}, \texttt{CONTROL\_STRUCTURE}\}$. As such, the inter-node value of a path is the proportion of nodes in the path which are considered 'target' nodes. We choose these `CALL` and `CONTROL_STRUCTURE` as target types because both function calls and control structures (e.g., `if-else` statements, `for` or `while` loops) are highly important structural elements that are relevant to the actual operation of a given code snippet. Therefore, token node pairs that are strongly related through these internal nodes are likely to be important to the snippet's representation. API and library calls are often considered to be "points-of-interest" by security researchers, as these calls often result in vulnerabilities through incorrect use, either directly (e.g., parameters are passed incorrectly into the call, the call is made at the wrong point in the code, etc.) or indirectly (e.g., parameters have been inadequately sanitised, etc.). Several previous studies, including [7], [20], [21], and [62], focus on function calls as key points in code. However, `CONTROL_STRUCTURE` nodes are integral to the structure of code snippets, and such nodes can be the difference between a vulnerable snippet and a safe one (e.g., a check to see if a buffer has space for a string before copying it in).

Chi-Squared heuristic: Selecting candidate path-contexts based on the correlation between a given path-context and the vulnerability or safety of functions it appears in. We calculate the correlation between a path-context's path (excluding the terminal nodes) and function vulnerability or safety using Cramér's V-test of association. First, we produce a vocabulary containing every path (excluding the terminal nodes) through every function in the dataset. Subsequently, we create a contingency table for each path. For a given path, let a be the number of safe functions containing the path, b be the number of vulnerable functions containing the path, c be the number of safe functions without the path, and d be the number of vulnerable functions without the path. We then calculate the correlation between a path-context and function vulnerability as follows:

$$\epsilon_{chi}(p) = \frac{(ad - bc)^2}{(a+b)(c+d)(a+c)(b+d)}$$

Let E be a set of pruning heuristics, $E = \{\epsilon_{random}, \epsilon_{ratio}, \epsilon_{inter-node}, \epsilon_{chi}\}$. A path pruner $\delta : 2^P \times E \times \mathbb{N} \mapsto 2^P$ is a function which takes a set of path-contexts $O \in 2^P$ and prunes it to a subset $Q \subseteq O$ of size $m \in \mathbb{N}$ by assigning a weight to each path-context according to a pruning heuristic $\epsilon \in E$ and selecting the m path-contexts with the highest weights.

D. EMBEDDING

Once the bags-of-path-contexts for each code snippet in the dataset have been pruned to the desired size, they are then used as input to a DL model to produce embeddings of length l. We adopt a path-attention model similar to code2vec. The embedder uses two vocabulary matrices: a node vocabulary $node\_vocab \in \mathbb{R}^{|X|\times d}$ and a path vocabulary $path\_vocab \in \mathbb{R}^{|Y|\times d}$ where X and Y are the sets of terminal nodes and the paths between terminal nodes, respectively, and d is the desired length for each node or path embedding. Although d could be given different values for the node and path embeddings, we set it to be the same for convenience. These vocabulary matrices are initialised randomly and learned during training.

For each path-context in a code snippet, a context vector

$$c = [node\_vocab_s; path\_vocab_j; node\_vocab_t] \in \mathbb{R}^{3d}$$

is produced where s, j, and t are the vocabulary indices for the path-context's start node, path, and end node respectively. Thus, the embedding for a path-context is produced by concatenating the embeddings of each of its constituent parts.

These context vectors are then fed into dense layers to produce combined context vectors $\tilde{c} \in \mathbb{R}^l$. For a path-context vector c_i, the combined context vector is calculated as follows:

$$\tilde{c}_i = \tanh(W \cdot c_i + b) \tag{2}$$

where l is the vector length, $W \in \mathbb{R}^{l\times 3d}$ is the weight matrix and b is the bias component. Both W and b are parameters to be learned in the dense layer. These combined context vectors are then aggregated into the vector $v \in \mathbb{R}^l$, as calculated in Eq. 3, to represent an input code snippet using an attention mechanism.

$$v = \sum_{i=1}^{m} \alpha_i \cdot \tilde{c}_i \tag{3}$$

v is the weighted sum of the combined context vectors of the selected m path contexts factored by their attention weights α_i which is calculated using Eq. 4, where v' is the current vector of the input code snippet, and W_α and b_α are the parameters to be learned in the attention module.

$$\alpha_i = \frac{exp(f(\tilde{c}_i, v'))}{\sum_{j=1}^{m} exp(f(\tilde{c}_j, v'))} \tag{4}$$

$$f(\tilde{c}_i, v') = \tanh\left(\mathrm{W}_\alpha \begin{bmatrix} \tilde{c}_i \\ v' \end{bmatrix} + \mathrm{b}_\alpha\right) \tag{5}$$

E. CLASSIFICATION

Finally, once the combined snippet vector v has been produced it is fed into a final dense layer with a single output which uses a sigmoid activation function to classify the input function as either vulnerable or safe. The binary cross-entropy loss of the model can be calculated by comparing this classification with the label associated with the input function and the model weights are updated.

IV. EXPERIMENTAL DESIGN

In this section we describe the experimental process used to evaluate SCEVD.

A. DATA PREPARATION

First, we obtained a dataset D of labelled vulnerable and safe code for testing. We chose to use the SARD Juliet test suite [63]; specifically CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer, CWE-399: Resource Management Errors, CWE-190: Integer Overflow or Wraparound, and CWE-400: Uncontrolled Resource Consumption.

SARD Juliet is a synthetic test suite of vulnerable and safe C/C++ code designed for testing and benchmarking VD tools. We focused on CWE-119, CWE-399, CWE-190, and CWE-400 because buffer, resource management, integer overflow, and resource exhaustion errors are vulnerabilities that are both common and can be extremely dangerous when exploited. In addition, a great deal of the existing VD literature also focuses on these vulnerability classes, making it easier to compare our results with those of previous studies. The sizes of the initial datasets are listed in Table 1.

TABLE 1. A breakdown of the sizes of all experimental datasets, including both classes, the total size of the dataset, and the size when balanced for classification.

Dataset	Vulnerable?		Totals	
	Yes	No	Unbalanced	Balanced
CWE-119	23,852	41,040	69,509	56,938
CWE-399	12,937	28,996	41,933	25,874
CWE-190	9,089	4,416	13,505	8,832
CWE-400	783	1,571	2,354	1,566

We then preprocessed these datasets to remove redundant information (e.g., code comments), and used Clang's Python bindings[4] to locate functions and variable names for normalisation (e.g., "`buffer`" became "`VAR_1`", "`memcpy`" became "`FUNC_1`", etc.). We then converted each function into an AST and a CPG using *Joern*.[5] For classification purposes we also balanced the datasets such that there were an equal number of safe and vulnerable test cases by randomly selecting a number of safe test cases equal to the total number of vulnerable test cases. Additionally, a 5-fold cross-validation evaluation has been conducted by splitting our datasets into training, validation, and testing data, with 80% of the data used for training, 10% for validation, and 10% for testing.

B. EXPERIMENTAL SETUP

Pathfinding: Once we produced the code graphs for our dataset we performed pathfinding on each graph to produce a full (i.e., unpruned) bag-of-path-contexts for each graph, including paths from the AST and the shortest paths from the CPG. As previously mentioned, for the CPG, we chose the shortest path between each pair of nodes as the one most representative of their relationship.

Path Pruning: For each graph, we pruned the bag-of-path-contexts to size $m = 200$ using a variety of pruning heuristics. The combination of graphs and pruning heuristics acted as the independent variables for our experiments. For both the AST and the CPG we used random, chi-squared pruning, and inter-node pruning. For the CPG we also used CPG-ratio pruning, which we could not apply to the AST due to its reliance on CPG edges to function.

We therefore test 7 models with different pruning heuristics and different input graphs:

- *AST_Random*: using a random pruner with ASTs as input data (baseline used in [2])
- *CPG_Random*: using a random pruner with CPGs as input data
- *AST_interNode*: using an inter-node pruner with ASTs as input data
- *CPG_interNode*: using an inter-node pruner with CPGs as input data
- *AST_chiSquared*: using a Chi-Squared pruner with ASTs as input data
- *CPG_chiSquared*: using a Chi-Squared pruner with CPGs as input data
- *CPG_ratio*: using a CPG-ratio pruner with CPGs as input data (this pruner cannot be applied to ASTs because the CPG-ratio for all AST-only paths will be 0).

Classification: Our classifier is the classification model described in Section III. Each model was trained to produce full code snippet vectors of length $l = 128$ using an Adam optimizer with a learning rate of 0.001, an epsilon of 1×10^{-7}, and a binary cross-entropy loss function. Our models and training pipeline were implemented using Keras[6] and TensorFlow.[7]

We trained our models for 100 epochs each on a workstation running Ubuntu 22.04 with a NVIDIA RTX A4500 GPU and an AMD Ryzen Threadripper PRO 5955WX CPU running at 4.0 GHz.

Metrics: The metrics used to evaluate our models were precision, recall, Area-Under-Curve (AUC), and F1-score. These values were calculated as follows:

$$Precision = \frac{\text{true positives}}{\text{true positives} + \text{false positives}}$$

$$Recall = \frac{\text{true positives}}{\text{true positives} + \text{false negatives}}$$

$$F1 = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

The AUC metric is the area under the Receiver Operating Characteristic (ROC). The ROC is a graph of the True Positive Rate (TPR, another name for the recall) against the False Positive Rate (FPR), which is calculated as follows:

$$FPR = \frac{\text{false positives}}{\text{false positives} + \text{true negatives}}$$

Thus the AUC is calculated as follows:

$$AUC = \int_{x=0}^{1} ROC(x)dx$$

where $ROC(x)$ is the TPR when x is the FPR.

C. RESEARCH QUESTIONS AND EXPERIMENTS

Our experiments seek to answer the following questions:

1) Does the use of heuristic-based pruning improve SCEVD's ability to predict code vulnerabilities compared to random pruning?
2) Does the use of CPGs improve SCEVD's ability to predict code vulnerabilities compared to the use of ASTs?

[4]https://pypi.org/project/clang/
[5]https://joern.io/

[6]https://keras.io/
[7]https://www.tensorflow.org/

3) Is SCEVD capable of identifying the path-contexts that are most relevant to the overall representation of a code graph?
4) Which of the pruning heuristics used by SCEVD is the most effective in consistently selecting path-contexts that are most relevant to the representation of the overall graph, and what properties of each dataset lend themselves to effective pruning by each heuristic?
5) How does SCEVD compare to other vulnerability detection models?

We construct the following series of experiments and analyses to test each of these research questions (RQs):

RQ1. In order to test the effectiveness of each pruning heuristic, we train and test each of the 7 models described in Section IVB. These models are classifiers which take a bag-of-path-contexts representing a the source code of a function as input and output a confidence value between 0 and 1 to indicate the probability that the given input is vulnerable. Classifications above a threshold of 0.5 are considered vulnerable and classifications below 0.5 are considered safe. These classifications can then be compared to the dataset labels to produce the metrics outlined in Section IVB. For each dataset and graph-type combination we then compare the results of the pruning heuristics in order to evaluate their performance against the baseline AST_random models.

RQ2. We use the classification metrics produced for RQ1 to address RQ2, however instead of examining the differences between models trained on different pruning heuristics and the same graph-types, we examine the differences between models trained on different graph-types and the same pruning heuristics.

RQ3. We extract the weight vector produced by the attention module for a given graph in order to examine the attention weights assigned to each path-context in the graph. We can then visualise these weights by using an example and assess their correctness based on our understanding of the vulnerabilities in the source code used to produce said graph.

RQ4. To evaluate the dataset properties which correlate with relatively stronger performances by each pruning heuristic, we calculate the variation in the weights produced by the chi-squared pruning heuristic between every path-context across each dataset. We then assess the differences in F1 scores between the models trained on datasets pruned by the inter-node heuristic and the chi-squared heuristic and correlate these with the variation in chi-squared weights. We choose to examine the chi-squared heuristic weights because the chi-squared heuristic targets correlations between functions rather than hard-coded properties of functions (as the inter-node heuristic does).

RQ5. Finally, to evaluate SCEVD's performance compared to other VD models and tools, we perform a comparative test on SCEVD with several other tools (VulSniper [10], DeepSim [57], and the Rough Auditing Tool for Security (RATS) static code analyser). We take the metrics for VulSniper and DeepSim reported by [10] as our baseline for comparison and test SCEVD under their experimental settings with the same datasets they used. We also produce our own replication of VulSniper in order to test the effectiveness of pruning heuristics on a different VD model. These experiments are also targeting function-level vulnerability detection, as with RQ1 and RQ2.

V. RESULTS AND EVALUATION

Our experiments are summarised by Table 2.[8] We address the research questions outlined in Section IVC as follows:

RQ1. We believe that these results demonstrate that heuristic-based pruning improves SCEVD's effectiveness as compared to random pruning. Random pruning has two major flaws: effectiveness and consistency. As discussed previously, for a given prediction made by SCEVD to be accurate, the graph's bag-of-path-contexts must contain at least some of the path-contexts that are relevant to the vulnerability of the function in question. This means that it is possible for random pruning to select all of the relevant path-contexts, some of them, or none of them; thus the F1 scores of SCEVD models trained on randomly pruned datasets can vary significantly.

The "Random (worst)" rows in Table 2 show the lowest F1 score obtained from models trained on repeated random pruning runs. All the models trained on randomly pruned data used the same random seed and were trained using the CPU only in order to make the training deterministic for the same input, while different random seeds were used for pruning the data. As such, the only possible cause of variation between models was the results of the random pruning. We found that the variance (best F1 – worst F1) in the scores achieved by random pruning runs could reach as high as 10.52% (for CWE-400's AST dataset) and was, on average, 4.52% across all datasets and graph types, while the average difference between the worst and mean F1 scores (worst F1 – mean F1) was -3.45%.

This inconsistency is problematic, especially given the problem domain of VD, as this variation between models casts doubt on any given prediction made by a model trained on randomly pruned data, and repeatedly randomly pruning and training models on the same dataset or prediction candidate is highly undesirable, wasting both time and resources. Conversely, heuristic-based pruners are completely deterministic and always select the same bags-of-path-contexts for the same dataset and graph type. Thus, for models trained deterministically, the variance over multiple runs with the same dataset, graph type, and pruner will always be 0%.

Additionally, we can observe that even when the average F1 scores of several random models are considered, models

[8]Note that each value for the Random (mean) rows is the independent average of that value across all random experiments; thus the AUC and F1 scores listed are not directly calculated using the listed precision and recall scores. The Random (worst) and Random (best) rows, however, contain values all taken from the same Random pruner experiment with the lowest and highest F1 scores for each Dataset/Graph pair, respectively. Additionally note that when we refer to the "best" or "worst" experiment we are referring to those with the highest or lowest F1 score, respectively.

TABLE 2. Classification metrics from each of the different pruning heuristics. Random experiments were run 10 times and the average, minimum, and maximum results are shown here. *Diff* denotes the percentage change between the mean and worst baseline model (AST_random) and the row's model.

Dataset	Graph	Pruner	Precision (%)	Recall (%)	AUC (%)	F1 (%)	Diff (mean)	Diff (worst)
CWE-119	AST	Random (mean)	88.00	88.75	94.21	88.29		
		Random (worst)	89.18	83.92	93.05	86.47	-1.82	
		Random (best)	85.09	93.23	94.40	88.97	0.68	2.50
		Inter-Node	**88.04**	**91.15**	**95.40**	**89.57**	1.28	3.10
		Chi-Squared	87.28	90.23	96.31	88.73	0.44	2.26
CWE-119	CPG	Random (mean)	85.89	90.31	94.38	88.56		
		Random (worst)	90.04	83.80	93.37	86.81	-1.75	
		Random (best)	88.28	90.40	94.67	89.33	0.77	2.52
		Inter-Node	85.41	94.84	95.44	89.87	1.58	3.40
		Chi-Squared	**85.89**	**95.92**	**96.77**	**90.63**	2.34	4.16
		CPG-Ratio	87.06	92.12	96.18	89.52	1.23	3.05
CWE-399	AST	Random (mean)	76.53	89.59	88.50	82.52		
		Random (worst)	74.27	89.13	87.14	81.03	-1.49	
		Random (best)	73.38	98.49	88.00	84.09	1.57	3.06
		Inter-Node	**76.77**	**95.37**	**90.09**	**85.07**	2.55	4.04
		Chi-Squared	76.48	93.37	91.37	84.09	1.57	3.06
CWE-399	CPG	Random (mean)	76.72	88.59	88.02	82.18		
		Random (worst)	74.71	88.74	86.67	81.13	-1.05	
		Random (best)	75.06	93.05	87.95	83.09	0.91	1.96
		Inter-Node	**76.86**	**93.40**	**88.62**	**84.33**	1.81	3.30
		Chi-Squared	76.46	92.25	90.80	83.61	1.09	2.58
		CPG-Ratio	80.71	82.40	90.15	81.55	-0.97	0.52
CWE-190	AST	Random (mean)	82.38	93.77	91.50	87.66		
		Random (worst)	82.90	85.92	91.65	84.38	-3.28	
		Random (best)	**81.58**	**99.65**	**90.95**	**89.71**	2.05	5.33
		Inter-Node	81.03	99.02	91.60	89.13	1.47	4.75
		Chi-Squared	81.99	96.91	92.94	88.83	1.17	4.45
CWE-190	CPG	Random (mean)	81.79	92.69	91.07	86.85		
		Random (worst)	83.61	87.00	91.10	85.27	-1.58	
		Random (best)	81.13	94.96	91.46	87.50	0.65	2.23
		Inter-Node	78.94	96.94	91.75	87.02	-0.64	2.64
		Chi-Squared	**84.34**	**95.59**	**93.42**	**89.61**	1.95	5.23
		CPG-Ratio	82.09	94.98	91.65	88.07	0.41	3.69
CWE-400	AST	Random (mean)	79.49	91.60	88.65	85.02		
		Random (worst)	69.50	83.36	82.37	75.80	-9.22	
		Random (best)	81.32	91.96	89.75	86.32	1.30	10.52
		Inter-Node	72.81	94.08	85.41	82.09	-2.93	6.29
		Chi-Squared	**77.75**	**97.24**	**91.94**	**86.41**	1.39	10.61
CWE-400	CPG	Random (mean)	81.18	88.71	87.89	84.70		
		Random (worst)	73.81	81.77	82.27	77.23	-7.47	
		Random (best)	78.68	93.06	86.99	85.27	0.57	8.04
		Inter-Node	72.83	90.63	89.73	80.76	-4.26	4.96
		Chi-Squared	**83.33**	**93.24**	**95.80**	**88.01**	2.99	12.21
		CPG-Ratio	70.44	91.63	87.03	79.65	-5.37	3.85
Average Differences						Random (worst)	-3.45	
						Random (best)	1.06	4.52

trained on randomly pruned datasets are still outperformed by those trained on heuristically pruned datasets, Across all datasets and both graph types, models trained on the best heuristic-based pruners outperformed those trained on randomly pruned datasets, demonstrating that even when the random pruner's worst performances are compensated for, it is unable to surpass the heuristic-based pruners' effectiveness. The best performing models trained on randomly pruned data were also usually outperformed by the models trained on the best heuristic-based pruner (only for CWE-190's AST dataset did the best random pruner perform better, and only by a very small margin of 0.58%). As mentioned previously, a given random pruning run could theoretically perform identically or better than a heuristic-pruning run, but this is infrequent. As such, owing to the inconsistency and comparatively poor performance of the random pruner, we believe that SCEVD is indeed improved through the use of heuristic-based pruning.

We experimented with several "combined-heuristic" pruners in which two of our non-random heuristics were combined ($\epsilon_{AB}(p) = \epsilon_A(p) \times \epsilon_B(p)$) where ϵ_{AB} is the combined heuristic, ϵ_A and ϵ_B are the two single heuristics, and p is the path-context. We found that the combined heuristic was typically very close to one of the heuristics used to create the combination. The best combined heuristic we used was an "inter-chi" heuristic, in which the normalised scores of the inter-node and chi-squared heuristics were multiplied together to produce the weight for a given path-context.

The largest difference between the inter-chi heuristic's F1 score and its constituent heuristics' F1 scores was for the CWE-399 AST test, with an F1 of 82.16%, which was 2.91% lower than that of the inter-node pruner. For the other three datasets, the disparities were smaller: the inter-chi pruner scored 1.50%, 2.34%, and 2.43% below the closest heuristic for each of the CPGs for CWE-119, CWE-190, and CWE-400, respectively. The smallest disparity we observed was between the inter-chi pruner and the inter-node pruner when applied to CWE-119's ASTs, for which the inter-node pruner performed 0.36% better than the inter-chi pruner. Thus, for all four datasets, the inter-chi heuristic performed worse than at least one of the inter-node or chi-squared heuristics, and was typically very close to one of them. We attribute this to the tendency of one of the two heuristics to "dominate" the selection process with slightly reduced effectiveness, while the other was mostly overlooked. We have chosen to exclude these results from our analysis as they complicate our data without providing further benefit or insight.

RQ2. To answer RQ2, we compare the changes in metric scores between the AST and CPG experiments for each of our datasets and each of our pruners. We exclude the CPG-ratio pruner as it cannot be used on ASTs. The results of this comparison are listed in Table 3.

TABLE 3. Change in metrics between AST and CPG graph types for each pruner/dataset pair. Change = CPG_metric - AST_metric.

Dataset	Pruner	AUC (%)	F1 (%)
CWE-119	Inter-Node	0.04	0.30
	Chi-Squared	0.46	1.90
	Random (mean)	0.17	0.27
CWE-399	Inter-Node	-1.47	-0.74
	Chi-Squared	-0.57	-0.48
	Random (mean)	-0.48	-0.34
CWE-190	Inter-Node	0.15	-2.11
	Chi-Squared	0.48	0.78
	Random (mean)	-0.43	-0.80
CWE-400	Inter-Node	4.31	-1.32
	Chi-Squared	3.87	1.59
	Random (mean)	-0.76	-0.32

We observe three trends from these changes.

1) Compared to ASTs, the use of CPGs usually results in a lower F1 score for the Inter-Node pruner. Only for CWE-119 does the F1 score improve between graph types for the Inter-Node pruner, and this difference is minimal ($< 1\%$).
2) Compared to ASTs, the use of CPGs usually results in a higher F1 score for the Chi-Squared pruner. Unlike the Inter-Node pruner, only for CWE-399 does the Chi-Squared pruner's F1 score suffer from the use of CPGs over ASTs, and this was also by $< 1\%$.
3) Compared to ASTs, the use of CPGs does not usually significantly impact the results of the random pruners. In no cases does the difference in results between graph types exceed 1% either positively or negatively.

Taken together these trends demonstrate that the effectiveness of the use of CPGs instead of ASTs depends on the pruner used for the experiments. The Inter-Node pruner is unaware of the kinds of edges found along a given path, as it uses only the properties of the nodes themselves to produce a weight for the path. As such, it does not make use of the CPG's more expressive edges. Since the paths in CPGs tend to be shorter than those for the AST (as CPGs draw "shortcuts" between distantly-related AST nodes), this means the Inter-Node pruner has less information with which to select important path-contexts and is therefore less discerning, leading to weaker classification results.

On the other hand, the Chi-Squared pruner *does* distinguish between CPG and AST path-contexts, as two paths passing through the same nodes but along different edges will appear different to the Chi-Squared pruner and will therefore be given different weights as they appear at different proportions among safe and vulnerable functions. Thus, when the types of edges in the CPG are even indirectly accounted for, the Chi-Squared pruner is able to produce better testing results than when it is applied to ASTs.

Finally, we see that for random pruning, there is no significant difference between using the AST or the CPG as the code graph. This leads us to conclude that the DL model used by SCEVD is *not* itself sensitive to the different graph types, however the pruning heuristics we test *are*. This lends further credence to our belief that intelligent pruning of path-contexts produces tangible results during training.

RQ3. In order to answer RQ3, we return to the motivating example outlined in **Section II-C**. One of the benefits of SCEVD is its explainability. It is possible to partially understand its operation by extracting the attention weights assigned to each path-context following training. Consider again the functions in Listings 3 and 4. As mentioned, the vulnerability in Listing 3 hinges on the existence of a meaningful relationship between the number of bytes allocated on the stack, pointed to by `data`, the maximum value of `i`, and the bytes accessed when a buffer of integers is indexed. When we examine the paths given the greatest attention by our SCEVD model, trained on the SARD Juliet CWE-119 dataset after Chi-Squared pruning, we can see that this relationship is indeed given a high priority. Figure 4 shows the top path-contexts selected by SCEVD.[9]

The most prominent path-contexts for Listing 3, respectively assigned roughly 20%, 17%, and 6% of the model's attention, describe this relationship. Key Path-Context (KPC) 1 runs between the `i` token in the loop comparison (`i < 10`) on line 9 and the `data` token that is assigned a pointer to the ten bytes of memory allocated on line 5. KPC 2 is between the `10` integer literal in the same loop comparison and the `data` token that is used as part of the index access on line 11. KPC 3 is between the variable type to which the allocated memory is cast (an integer pointer), and the `i` token on line 11 used to access indexes in `data`.

[9]For the sake of readability only path-contexts with an attention weight greater than 5% are shown, all CFG and PDG edges that are not a part of a path have been removed, and nodes that are not relevant to any of the path-contexts have been collapsed into the node representing their line in the code. The width of the line for each path represents the weight attributed to the path-context by the attention module. Also note that where two edges with the same label connect the same node pair, the edge in question is the same edge but is shown for each path of which it is a member, hence the duplication.

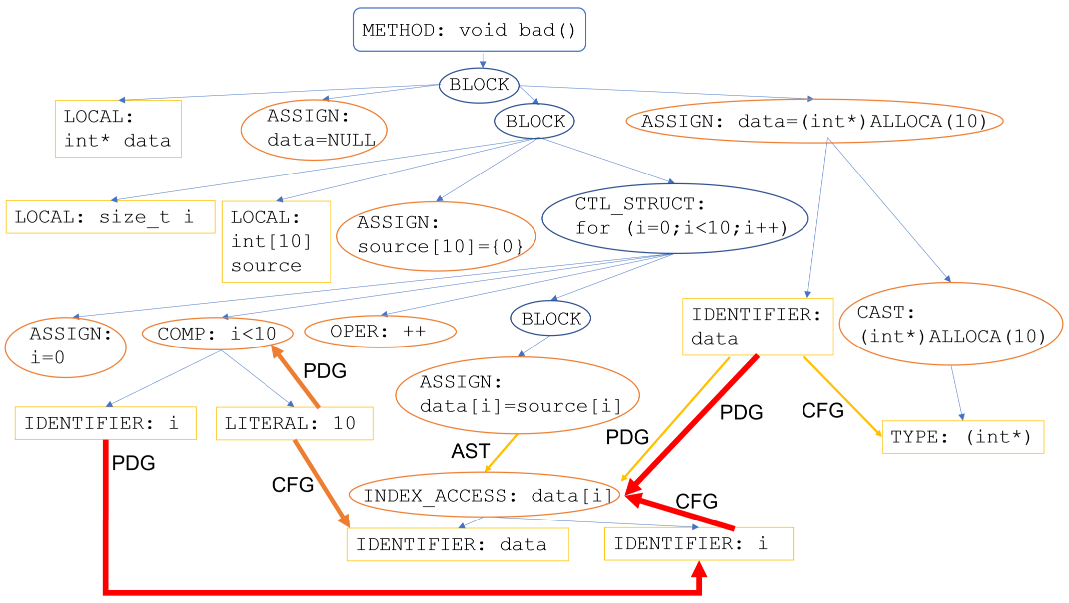

FIGURE 4. **Condensed CPG and path-context attentions selected for Listing 3. Unlabelled edges are AST edges.**

These three KPCs are assigned nearly half of SCEVD's total attention for this function; no other path-contexts are given an attention weight higher than 5% and only 18 of the 200 path-contexts are given an attention weight greater than 1%.

This assignment of attention shows that SCEVD is indeed capable of selecting the most relevant path-contexts for the vulnerability of this function. KPC 1 shows that `i` is used to access indexes in a buffer, KPC 2 shows the relationship between the maximum value of `i` (10) and the actual command to access an integer in said buffer, and KPC 3 shows that the buffer that is accessed is an integer buffer. This third relationship is relevant because different variable types in C have different sizes. Were `data` a character or short integer pointer, no vulnerability would be present, as the `char` or `short` types in C are only a single byte long, so accessing up to an index of 9 would be safe. However, as `data` is an integer pointer and is therefore 4 bytes long, a buffer overflow exists in this code.

Additionally, it should be noted that very few of the most relevant path-contexts address the `source` buffer at all. This is because neither the contents nor the size of `source` impact the vulnerability in this function. The size of `source` would be relevant if an alternative means of copying its contents to `data` were used (like `strcpy` or `memcpy`), or if the maximum size of `i` were the result of a `strlen(source)` call. An Unauthorised Code Execution Attack would also be possible if the contents of `source` were somehow tailored to be malicious. However, even without these additions, a vulnerability is still present in Listing 3, as memory corruption will occur regardless of the specific values being copied into the unallocated memory.

An examination of the attention given to path-contexts in order to produce a vulnerability prediction shows that SCEVD's predictions can be explained and are in line with human assessments of code vulnerability. We believe this serves as a good example of SCEVD's capabilities, that repeat across the rest of our datasets, as evidenced by SCEVD's good prediction results shown in Table 2.

Additionally, this explainability is valuable in itself, as SCEVD is not only capable of predicting vulnerabilities, but also of explaining why it made the predictions it made. The desire for e**X**plainable **A**rtificial **I**ntelligence (XAI) has been growing as the capabilities of existing AI systems grow, and SCEVD's ability to explain parts of its decision making is a step in this direction. This can be used by security practitioners and software engineers in order to pinpoint the specific sections of their source code in which vulnerabilities are present.

RQ4. When examining the relative differences in performance between each of the heuristic-based pruners, we note two significant facts. The first is that the CPG-ratio pruner consistently produced the worst results of all three heuristic-based pruners (though it still outperformed the worst-case Random pruner for every dataset). The second is that the results from models trained on the Inter-Node and Chi-Squared pruners do not produce a clean and unambiguous "winner" for every dataset and every graph type.

Regarding the performance of the CPG-ratio pruner, these results can be explained by the fact that it is a fairly naive pruner that does not use a great deal of information to make its selections. The CPG-ratio pruner relies only on the semantic qualities of the relationship between node pairs and does not consider the contents of the terminal nodes or those of the nodes themselves.

Regarding the relative strengths of Chi-Squared pruning and Inter-Node pruning, neither pruner can be considered decisively "better" than the other in all cases, though it should be noted that for the models trained on the CPGs of both of the larger datasets (CWE-119 and CWE-399, with 56,938 and 25,874 functions respectively, after class balancing), the differences in F1 scores between the two pruners are very small (0.76% and -0.72%, respectively), while for the smaller datasets (CWE-190 and CWE-400, with 8,832 and 1,566 functions respectively, after class balancing) the Chi-Squared pruner produced significantly better results than the Inter-Node pruner (with a difference of 2.59% and 7.25%, respectively). This is likely because path-contexts that are strongly correlated with a vulnerability of one type may appear in functions that are safe for another type of vulnerability. The CWE-119 and CWE-399 datasets contain many sub-classes of vulnerability, while the CWE-190 and CWE-400 datasets do not, leading the Chi-Squared pruner to be more sensitive to path-contexts correlated with vulnerabilities for single-class vulnerabilities and less sensitive to correlations among families of vulnerabilities.

TABLE 4. Difference between Chi-Squared pruning F1 score and Inter-Node pruning F1 score for each dataset/graph-type paired with the standard deviation of the distribution of Chi-Squared values for each dataset/graph-type pair.

Dataset	Graph	F1 Difference (%)	σ
CWE-119	AST	-0.84	0.0015
	CPG	0.76	0.0010
CWE-399	AST	-0.98	0.0010
	CPG	-0.72	0.0009
CWE-190	AST	-0.30	0.0019
	CPG	2.59	0.0020
CWE-400	AST	4.32	0.0070
	CPG	7.25	0.0051

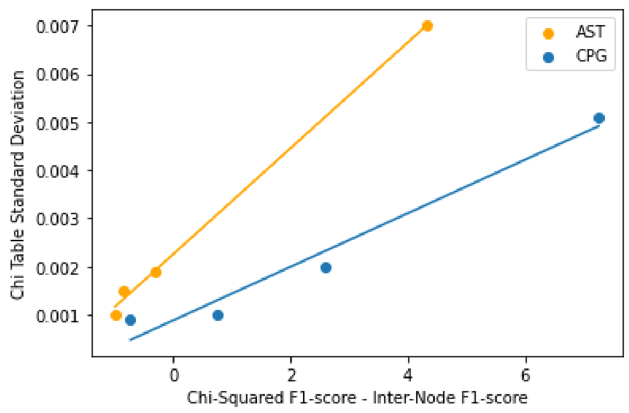

FIGURE 5. Correlation between Chi-Squared and Inter-Node pruning F1 scores for each dataset/graph-type pair and the standard deviation of the distribution of Chi-Squared values for each dataset/graph-type pair.

This phenomenon can be observed when comparing the differences in F1 score between models trained on each pruner with the standard deviation of the distribution of path-context correlations in each dataset. Table 4 (charted in Figure 5) shows that as the standard deviation of the path-context correlations in a dataset increases, so too does the difference in F1 score for that dataset between models trained on the Chi-Squared pruner and the Inter-Node pruner. For datasets with a narrow range of chi values for each path context and therefore fewer strongly correlated path-contexts, the Chi-Squared pruner is less able to discriminate between KPCs and uncorrelated path-contexts than for those with a wide range of chi values.

As a result, we find that for small datasets, or those with a relatively low variation of vulnerability types, the Chi-Squared pruner is more appropriate as the correlations between particular path-contexts and the presence or absence of vulnerabilities is considerably stronger, whereas for larger or more varied datasets, the Inter-Node pruner performs better.

RQ5. Finally, we can compare SCEVD to other vulnerability detection tools. It should be noted that our focus in this work is on graph pruning and feature selection, which has not (to our knowledge) been explored in other graph-based vulnerability discovery works. While pruning input data is an inherent component of any VD model, as all such models must limit their input to a fixed size, this is typically not the focus of other works. Other architectures may strongly contribute to the performance of a given model, however our novelty consists of the pruning heuristics we propose and employ, not in our model architecture.

Nonetheless, we can compare the path-attention model to some existing graph-based code embedders for vulnerability discovery in Table 5 [10], [57], as well as the RATS[10] static analysis tool.[11] For each dataset we compare our best pruner's results for the CPG with the results provided by the tool with which we are comparing ourselves.

TABLE 5. Comparison of SCEVD's metrics and metrics of other similar tools. P and R stand for Precision and Recall respectively. Change refers to the difference in F1 score between SCEVD and the compared VD tool (SCEVD F1 - Comparison F1). SCEVD's results were obtained using [10]'s experimental settings (train-test-val split of 6:2:2) and are thus different to the results found in Table 2. VulSniper ([10]) is the results found in [10], while VulSniper$_{Rep}$ is our replication of [10]'s VulSniper experiments.

Dataset	Model	P (%)	R (%)	F1 (%)	Change (F1 %)
CWE-119	**SCEVD** (ϵ_{chi})	**82.81**	**94.16**	**88.12**	
	VulSniper ([10])	88.70	73.80	80.60	7.52
	VulSniper$_{Rep}$	66.76	67.34	67.05	21.07
	DeepSim	71.60	58.40	64.40	23.72
	RATS	77.54	37.93	50.94	37.18
CWE-399	**SCEVD** (ϵ_{inter})	**81.04**	**94.49**	**87.25**	
	VulSniper ([10])	80.40	67.30	73.30	13.95
	VulSniper$_{Rep}$	76.07	60.68	67.51	19.74
	DeepSim	77.20	47.80	59.10	28.15
	RATS	64.06	28.14	39.10	48.15
CWE-190	**SCEVD** (ϵ_{chi})	**83.15**	**90.28**	**86.57**	
	VulSniper$_{Rep}$	88.86	77.63	82.87	3.70
	RATS	66.98	15.86	25.65	60.92
CWE-400	**SCEVD** (ϵ_{chi})	**74.72**	**91.13**	**82.11**	
	VulSniper$_{Rep}$	82.81	55.21	66.25	15.86
	RATS	67.74	41.31	51.32	30.79

Results for "VulSniper ([10])" and DeepSim were drawn from Duan et al. [10], who did not use CWE-190 or CWE-400 for their experiments, and as such we cannot compare them to SCEVD for those datasets. VulSniper is another tool which

[10]https://github.com/andrew-d/rough-auditing-tool-for-security
[11]We used the default warning level $w = 2$ for RATS.

requires that code graphs be pruned to a fixed size (in their case 128 nodes, as they do not use paths through the graph for learning). Their pruning heuristic involves selecting the nodes which are textually near to statements including API calls. Though it is not clear precisely how this is done, this heuristic does not take advantage of graph information (only textual information) and so could be considered analogous to a compromise between a inter-node pruner, as it targets a particular node type, and random pruning, as textual proximity is not a reliable metric for vulnerability-relevance (given that textual proximity is often not correlated with graphical proximity). DeepSim [57], much like code2vec, was not designed for vulnerability discovery, however it uses CFGs and data flow graphs (DFGs) to learn from code. The results shown for DeepSim were produced by Duan et al. [10]. DeepSim's pruning method for oversized functions is to discard DFG information over than the maximum input size.

As [10] does not supply source code, "VulSniper$_{Rep}$" is our attempt to replicate the model architecture and node pruning used in [10], however there are differences between our implementation and that in [10] due to incomplete details about [10]'s implementation.[12] Regardless, we find that SCEVD outperforms VulSniper for every dataset, both our replication and [10]'s reported performance, as well as DeepSim.

We also compare our results to the RATS static analysis tool. RATS does not require a fixed-length input and thus does not perform code graph pruning, however we include it as a point of comparison between SCEVD and a non-ML-based VD tool. As Table 5 shows, SCEVD outperforms RATS by a considerable margin.

Based on the results shown in Table 5, we can see that SCEVD noticeably improves over VulSniper, DeepSim, and RATS. This is likely due to a combination of factors, including improvement in pruning heuristics and in model architecture.

For further comparison, and in order to minimise the impact of our path-attention model architecture on our results, we also tested various pruning heuristics on the VulSniper model architecture. Because VulSniper takes nodes rather than path-contexts as input, we could not use identical pruning heuristics to those used for the path-attention model. The heuristics we used were as follows:

- A random heuristic:

$$\epsilon_{random}(n) = \text{rand}(n)$$

[12]Our replication uses a tensor of shape 128 × 128 × 104 instead of 128 × 128 × 144, as [10] uses tensors with manually selected node features which were not comprehensively listed, nor was a citation for the standard they used given. Additionally, the number of 1D convolutional filters and kernel size in the attention module, and the number of hidden units in the dense classification module are not listed and must be assumed (we use 32, 3, and 64 for these, respectively). We also attempt to replicate their node pruning method as described, though we cannot be sure that our replication is identical to [10].

- A replication of [10]'s heuristic, which we call "Call-Proximity":

$$\epsilon_{call-prox}(n) = |\text{line}(key\_node) - \text{line}(n)|$$

  where line(n) is the first line in the text of the code containing the structure that node n refers to.
- A "Property" heuristic, which prioritises CALL and CONTROL_STRUCTURE nodes:

$$\epsilon_{property}(n) = \begin{cases} 1 & \text{if } node\_type(n) \in target\_types \\ 0 & \text{else} \end{cases}$$

- An approximation of our inter-node heuristic, which first selects CALL and CONTROL_STRUCTURE nodes and then ranks nodes based on the number of CALL and CONTROL_STRUCTURE neighbours they share:

$$\epsilon_{inter-node} = \begin{cases} \infty & \text{if } node\_type(n) \in \\ & target\_types \\ target\_neighbours(n) & else \end{cases}$$

  where $target\_neighbours(n)$ is the number of nodes adjacent to n whose type is in $target\_types$.

For both $\epsilon_{property}$ and $\epsilon_{inter-node}$, $target\_nodes$ = $\{\texttt{CALL}, \texttt{CONTROL\_STRUCTURE}\}$

TABLE 6. Classification metrics for each different pruning heuristic applied to VulSniper.

Dataset	Pruner	P (%)	R (%)	AUC (%)	F1 (%)
CWE-119	Random	60.94	54.10	63.11	57.32
	Call-Proximity	66.76	67.34	72.50	67.05
	Property	68.70	56.83	69.98	62.20
	Inter-Node	**78.60**	**66.04**	**79.82**	**71.77**
CWE-399	Random	60.81	47.95	58.45	53.62
	Call-Proximity	76.07	60.68	73.01	67.51
	Property	73.82	63.94	72.07	68.53
	Inter-Node	**73.24**	**70.91**	**78.15**	**72.06**
CWE-190	Random	68.75	63.51	70.43	66.03
	Call-Proximity	88.86	77.63	90.42	82.87
	Property	86.49	75.79	86.34	80.79
	Inter-Node	**91.38**	**78.16**	**92.47**	**84.26**
CWE-400	Random	61.00	64.21	64.68	62.56
	Call-Proximity	82.81	55.21	80.13	66.25
	Property	69.23	66.32	76.47	67.74
	Inter-Node	**73.47**	**77.42**	**82.42**	**75.39**

Table 6 shows the results of our experimentation using the VulSniper model architecture and the above pruning heuristics. We find that, as with the path-attention model, randomly pruning nodes produces the worst results for every dataset. [10]'s call-proximity heuristic and the property heuristic both involve some amount of randomness, either when ties occur between nodes (which is guaranteed for $\epsilon_{property}$) or due to the weaker relationship between textually proximal nodes (for $\epsilon_{call-prox}$). The inter-node heuristic is the most attentive to node properties and relationships out of the heuristics used, and consequently inter-node models produce the best metrics of the pruning heuristics tested. Thus, we find that even for models with different architectures, heuristic-based code graph pruning is most effective.

VI. RELATED WORK

A. DEEP LEARNING FOR VULNERABILITY DISCOVERY

Many studies have focused on applying deep learning to vulnerability discovery, and a variety of approaches to the problem exist.

Token-based Code Embedding involves treating code as a sequence of tokens, and so the DL models used are those tailored for sequential data. Recurrent Neural Networks (RNNs) and their variations are a class of sequential processing DL architectures. Dam et al. and Russel et al. [9], [31] are two early and significant works in token-based code embedding for VD. Russel et al. uses both RNNs and Convolutional Neural Networks (CNNs) as options for learning from code, while Dam et al. uses Long Short Term Memory (LSTM). Li et al. [20] also used LSTMs (specifically bi-directional LSTMs, or BLSTMs) as well as proposing the *code gadget*, a series of code statements related to a key statement (usually an library/API function call). This work was then built on by [21], [62], and [22], using either BLSTMs or bi-directional RNNs (BRNN).

For token embedding [31] uses a conventional embedding layer for this task and [20], [62] use the *word2vec* [25] algorithm.

As the popularity of transformer-based [34] models for natural language processing has grown, they have also been applied to VD, including [13], [14], [50] who all use the CodeBERT [12] architecture.

Wu et al. [42] note that some of the weaknesses of token-based code embedding include difficulty in selecting only relevant code for vulnerability detection, particularly that "code slicing ...may not cover all the vulnerable code snippets". This problem, as well as the lack of structural information in token-based code embedding [42], [48] are major limitations. However, token-based code embeddings do not require the full context of the function to be used (unlike code graphs, which require completely parsable code to be produced) and can leverage recent advances in NLP.

Graph-Based Code Embedding for VD has seen a wide variety of DL architectures used. Some architectures operate directly on graphs, while other works (like ours) convert graphs into structures which can be used by other model architectures.

Graph Neural Networks (GNNs) are a family of neural networks designed for graphical input and take inspiration from CNNs. They learn by updating node representations using adjacent nodes, and variations like Recurrent Graph Neural Networks (RGNNs, or GRNNs) and Gated Graph Neural Networks (GGNN) add residual connections. Works using Graph Neural Networks (GNNs) and their variants include [5], [6], [19], [27], [28], [29], [32], [33], [41], [56], [59], [61], [61]. Graph Attention Networks (GATs) [35] are used by [16] and [18], which include an attention mechanism to evaluate the importance of neighbouring nodes to one another.

Bilgin et al. [3] converts ASTs into binary trees which are then flattened into a sequence of nodes up to a set depth. These are used as input to a CNN before classification. VulCNN [44] likewise uses a CNN for graph learning, however they first represent nodes in a format conceptually similar to images. DeepVulSeeker [37] combines both token representations produced by a pre-trained transformer model and graphical information using a self-attention head before classification. VulSniper [10] represents graph nodes based on their features (including node type, variable type, adjacency to other nodes, etc.) and uses a deep neural network with an attention mechanism for VD.

A notable limitation of graph-based vulnerability discovery is node-embedding [42]. Graph-based models require meaningful node embeddings in order to learn, but embedding these nodes can be difficult. Some works embed nodes using the word embeddings of the tokens they represent [5], [28], [29], [59], [61] which are produced by pre-trained word embedding models (word2vec, doc2vec, BERT). Because of this, these word embeddings are not trained to represent tokens in the context of vulnerability detection, limiting their effectiveness. Other works like VulSniper [10] use tailored node embeddings which rely on expert knowledge, while code2vec [2] learns node and path embeddings during training which are thus trained based on the downstream task. As this work uses the path-attention model in [2], our node and path embeddings are trained on vulnerability detection.

B. OTHER CODE GRAPH LEARNING TASKS

In addition to VD, code graph learning has been applied to a variety of other code comprehension problems.

Code clone detection is the task of identifying semantically similar or identical source code. Haridas et al and Wang et al. [15], [38] use GCNs for clone detection, while [52], [53], [55] use AST-based variants of LSTMs and BRNNs respectively. DeepSim [57] uses a method similar to [10] of representing nodes using selected features.

Zhang et al. [52] also applies graph learning to classifying source code by task. Likewise, Zhang et al. [54] uses a variant of the transformer encoder designed to use graphs as input, also performs code classification, as well as method name prediction. GNNs are employed for method name prediction [26], as is the path-attention model [2]. Related to name prediction, [23], [40], [60] all use graph learning for automatic comment generation, all using LSTMs [40], [60] and GRUs [23] applied to source ASTs. In close to a reverse case, [36] applies LSTMs to ASTs for source code retrieval, which takes natural language input and produces a code fragment from an existing code repository.

CNNs and LSTMs have been applied to CFGs in order to detect bugs in source code [17], as have GNNs [39]. Bug detection is related to VD but not identical in that a bug generally refers to any undesirable or unintended mistake in source code, while a vulnerability is specifically those which result in weaknesses in a systems security [64]. Finally, Ramadan et al. [30] uses ASTs as input to LSTMs in order to predict code performance for a selection of computational problems.

VII. LIMITATIONS

This study has some potential limitations, primarily the quality of the SARD Juliet dataset. We chose to use the SARD Juliet dataset because it is reliably labeled and very easy to work with, however it has some drawbacks. Juliet is a synthetic dataset and was generated following a list of patterns for vulnerable code and a list of variations, often minor, on the generated code. Both vulnerable and non-vulnerable versions of most entries are produced. The primary advantage of this is that the labels for each function are guaranteed to be reliable, however the disadvantage of this method is that Juliet has a low uniqueness between entries (as many entries have very small differences with one another) and that the entries in Juliet exist only to be examples of vulnerable or safe code. Unlike realistic datasets (i.e., Big-Vul [11], Devign [59], and D2A [58]), whose entries have some other purpose and are reflective of code found ''in the wild'', entries in Juliet are typically fairly minimal. This presents a limitation to our model's application to more complex datasets.

However, the limitations of other datasets are equally problematic and lead to our selection of Juliet for this paper. Big-Vul and Devign were both created by scraping Git commits from open source repositories both before and after security patches were added, and Devign was subsequently manually validated. D2A is a collection of open source code which was labelled using a static analysis tool.

Big-Vul is a very large dataset of real C/C++ code, however Croft et al. [8] find that its accuracy over a sample they examined was only 54.3%. We found it had issues with completeness when we attempted to produce code graphs for its entries, as C/C++ cannot be completely parsed without full context which Big-Vul does not provide. Devign is more accurate than Big-Vul, however it is also considerably smaller, limiting its usefulness for training VD models. Finally, D2A had a sampled accuracy of 28.6%, which is far too low to be usable for training, likely due the high false-positive rate of static analysis tools.

VIII. CONCLUSION AND FUTURE WORK

In this paper we present SCEVD, an ML-based tool for automated vulnerability detection that utilises semantic information and heuristic-based pruning in order to select features from code graphs that are most relevant to the presence or absence of vulnerabilities in source code. We examine a set of pruning heuristics and determine that informed path-context selection produces better results than the previously random process. We find that when pruning heuristics leverage the additional semantic information provided by CPGs, particulrly relationship-property information, they perform better than when that information is neglected. We find that SCEVD's use of path-context attentions provides a partial explanation for its classifications, and we find the dataset properties that correlate with the relative strengths and weaknesses of each of our pruning heuristics. Finally, we compare SCEVD to some existing VD models and tools and find that both SCEVD's model architecture and pruning heuristics lead to better performance.

There are ways in which future work could expand or improve our findings. While SCEVD uses bags-of-path-contexts for learning, the principle of pruning code graphs before training could be extended to other types of input data and other ML architectures, such as GNNs or GATs. It is also possible that other pruning heuristics, or those more targeted at specific vulnerability classes, could produce better results. Finally, work could be done to extend SCEVD's explainablility, as it is presently only able to highlight path-contexts that are *relevant* to a program's representation, rather than those that explicitly indicate vulnerabilities.

ACKNOWLEDGMENT

The authors would like to thank Tata Consultancy Services Ltd. (TCS) for co-funding this project and Debi Prasad Pati from TCS for comments and discussions.

REFERENCES

[1] M. Allamanis, M. Brockschmidt, and M. Khademi, ''Learning to represent programs with graphs,'' 2017, *arXiv:1711.00740*.

[2] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, ''code2vec: Learning distributed representations of code,'' *Proc. ACM Program. Lang.*, vol. 3, pp. 1–29, Jan. 2019, doi: 10.1145/3290353.

[3] Z. Bilgin, M. A. Ersoy, E. U. Soykan, E. Tomur, P. Çomak, and L. Karaçay, ''Vulnerability prediction from source code using machine learning,'' *IEEE Access*, vol. 8, pp. 150672–150684, 2020, doi: 10.1109/ACCESS.2020.3016774.

[4] S. Cao, X. Sun, L. Bo, Y. Wei, and B. Li, ''BGNN4VD: Constructing bidirectional graph neural-network for vulnerability detection,'' *Inf. Softw. Technol.*, vol. 136, Aug. 2021, Art. no. 106576, doi: 10.1016/j.infsof.2021.106576.

[5] S. Cao, X. Sun, L. Bo, R. Wu, B. Li, and C. Tao, ''MVD: Memory-related vulnerability detection based on flow-sensitive graph neural networks,'' in *Proc. IEEE/ACM 44th Int. Conf. Softw. Eng. (ICSE)*, May 2022, pp. 1456–1468, doi: 10.1145/3510003.3510219.

[6] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, ''Deep learning based vulnerability detection: Are we there yet?'' *IEEE Trans. Softw. Eng.*, vol. 48, no. 9, pp. 3280–3296, Sep. 2022, doi: 10.1109/TSE.2021.3087402.

[7] X. Cheng, H. Wang, J. Hua, G. Xu, and Y. Sui, ''DeepWukong: Statically detecting software vulnerabilities using deep graph neural network,'' *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 3, pp. 1–33, Jul. 2021, doi: 10.1145/3436877.

[8] R. Croft, M. A. Babar, and M. M. Kholoosi, ''Data quality for software vulnerability datasets,'' in *Proc. IEEE/ACM 45th Int. Conf. Softw. Eng. (ICSE)*, May 2023, pp. 121–133, doi: 10.1109/ICSE48619.2023.00022.

[9] H. Khanh Dam, T. Tran, T. Pham, S. W. Ng, J. Grundy, and A. Ghose, ''Automatic feature learning for vulnerability prediction,'' 2017, *arXiv:1708.02368*.

[10] X. Duan, J. Wu, S. Ji, Z. Rui, T. Luo, M. Yang, and Y. Wu, ''VulSniper: Focus your attention to shoot fine-grained vulnerabilities,'' in *Proc. 28th Int. Joint Conf. Artif. Intell.*, Macao, China, Aug. 2019, pp. 4665–4671, doi: 10.24963/IJCAI.2019/648.

[11] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, ''A C/C++ code vulnerability dataset with code changes and CVE summaries,'' in *Proc. IEEE/ACM 17th Int. Conf. Mining Softw. Repositories (MSR)*. New York, NY, USA: Assoc. Comput. Machinery, May 2020, pp. 508–512, doi: 10.1145/3379597.3387501.

[12] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, ''CodeBERT: A pre-trained model for programming and natural languages,'' 2020, *arXiv:2002.08155*.

[13] M. Fu and C. Tantithamthavorn, ''LineVul: A transformer-based line-level vulnerability prediction,'' in *Proc. IEEE/ACM 19th Int. Conf. Mining Softw. Repositories (MSR)*, May 2022, pp. 608–620, doi: 10.1145/3524842.3528452.

[14] H. Hanif and S. Maffeis, "VulBERTa: Simplified source code pre-training for vulnerability detection," in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, Jul. 2022, pp. 1–8, doi: 10.1109/IJCNN55064.2022.9892280.

[15] P. Haridas, G. Chennupati, N. Santhi, P. Romero, and S. Eidenbenz, "Code characterization with graph convolutions and capsule networks," *IEEE Access*, vol. 8, pp. 136307–136315, 2020, doi: 10.1109/ACCESS.2020.3011909.

[16] D. Hin, A. Kan, H. Chen, and M. A. Babar, "LineVD: Statement-level vulnerability detection using graph neural networks," in *Proc. IEEE/ACM 19th Int. Conf. Mining Softw. Repositories (MSR)*, May 2022, pp. 596–607, doi: 10.1145/3524842.3527949.

[17] X. Huo, M. Li, and Z.-H. Zhou, "Control flow graph embedding based on multi-instance decomposition for bug localization," in *Proc. AAAI Conf. Artif. Intell.*, Apr. 2020, vol. 34, no. 4, p. 4, Apr. 2020, doi: 10.1609/aaai.v34i04.5844.

[18] M. Li, C. Li, S. Li, Y. Wu, B. Zhang, and Y. Wen, "ACGVD: Vulnerability detection based on comprehensive graph via graph neural network with attention," in *Information and Communications Security* (Lecture Notes in Computer Science), D. Gao, Q. Li, X. Guan, and X. Liao, Eds. Cham, Switzerland: Springer, 2021, pp. 243–259, doi: 10.1007/978-3-030-86890-1_14.

[19] Y. Li, S. Wang, and T. N. Nguyen, "Vulnerability detection with fine-grained interpretations," in *Proc. 29th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, Aug. 2021, pp. 292–303, doi: 10.1145/3468264.3468597.

[20] 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," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2018, pp. 1–12, doi: 10.14722/ndss.2018.23158.

[21] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "SySeVR: A framework for using deep learning to detect software vulnerabilities," *IEEE Trans. Dependable Secure Comput.*, vol. 19, no. 4, pp. 2244–2258, Jul. 2022, doi: 10.1109/TDSC.2021.3051525.

[22] Z. Li, D. Zou, S. Xu, Z. Chen, Y. Zhu, and H. Jin, "VulDeeLocator: A deep learning-based fine-grained vulnerability detector," *IEEE Trans. Dependable Secure Comput.*, vol. 19, no. 4, pp. 2821–2837, Jul. 2022, doi: 10.1109/TDSC.2021.3076142.

[23] Y. Liang and K. Zhu, "Automatic generation of text descriptive comments for code blocks," in *Proc. AAAI Conf. Artif. Intell.*, Apr. 2018, vol. 32, no. 1, p. 1, doi: 10.1609/aaai.v32i1.11963.

[24] G. Lin, S. Wen, Q.-L. Han, J. Zhang, and Y. Xiang, "Software vulnerability detection using deep neural networks: A survey," *Proc. IEEE*, vol. 108, no. 10, pp. 1825–1848, Oct. 2020, doi: 10.1109/JPROC.2020.2993293.

[25] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," 2013, *arXiv:1301.3781*.

[26] L. Liu, H. Nguyen, G. Karypis, and S. Sengamedu, "Universal representation for code," in *Advances in Knowledge Discovery and Data Mining* (Lecture Notes in Computer Science), K. Karlapalem, H. Cheng, N. Ramakrishnan, R. K. Agrawal, P. K. Reddy, J. Srivastava, and T. Chakraborty, Eds. Cham, Switzerland: Springer, 2021, pp. 16–28, doi: 10.1007/978-3-030-75768-7_2.

[27] H. H. Nguyen, N.-M. Nguyen, H.-P. Doan, Z. Ahmadi, T.-N. Doan, and L. Jiang, "MANDO-GURU: Vulnerability detection for smart contract source code by heterogeneous graph embeddings," in *Proc. 30th ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.* New York, NY, USA, Nov. 2022, pp. 1736–1740, doi: 10.1145/3540250.3558927.

[28] H. V. Nguyen, J. Zheng, A. Inomata, and T. Uehara, "Code aggregate graph: Effective representation for graph neural networks to detect vulnerable code," *IEEE Access*, vol. 10, pp. 123786–123800, 2022, doi: 10.1109/ACCESS.2022.3216395.

[29] V.-A. Nguyen, D. Q. Nguyen, V. Nguyen, T. Le, Q. H. Tran, and D. Phung, "ReGVD: Revisiting graph neural networks for vulnerability detection," 2021, *arXiv:2110.07317*.

[30] T. Ramadan, T. Z. Islam, C. Phelps, N. Pinnow, and J. J. Thiagarajan, "Comparative code structure analysis using deep learning for performance prediction," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, Mar. 2021, pp. 151–161, doi: 10.1109/ISPASS51385.2021.00032.

[31] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley, "Automated vulnerability detection in source code using deep representation learning," in *Proc. 17th IEEE Int. Conf. Mach. Learn. Appl. (ICMLA)*, Dec. 2018, pp. 757–762, doi: 10.1109/ICMLA.2018.00120.

[32] L. Šikic, A. S. Kurdija, K. Vladimir, and M. Šilic, "Graph neural network for source code defect prediction," *IEEE Access*, vol. 10, pp. 10402–10415, 2022, doi: 10.1109/ACCESS.2022.3144598.

[33] W. Tang, M. Tang, M. Ban, Z. Zhao, and M. Feng, "CSGVD: A deep learning approach combining sequence and graph embedding for source code vulnerability detection," *J. Syst. Softw.*, vol. 199, May 2023, Art. no. 111623, doi: 10.1016/j.jss.2023.111623.

[34] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems*. Red Hook, NY, USA: Curran Associates Inc., Jun. 2017, pp. 6000–6010.

[35] P. VeliÄkoviÄ, G. Cucurull, A. Casanova, A. Romero, P. LiÃ, and Y. Bengio, "Graph attention networks," 2018, *arXiv:1710.10903*.

[36] Y. Wan, J. Shu, Y. Sui, G. Xu, Z. Zhao, J. Wu, and P. Yu, "Multi-modal attention network learning for semantic source code retrieval," in *Proc. 34th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2019, pp. 13–25, doi: 10.1109/ASE.2019.00012.

[37] J. Wang, H. Xiao, S. Zhong, and Y. Xiao, "DeepVulSeeker: A novel vulnerability identification framework via code graph structure and pre-training mechanism," 2022, *arXiv:2211.13097*.

[38] W. Wang, G. Li, B. Ma, X. Xia, and Z. Jin, "Detecting code clones with graph neural network and flow-augmented abstract syntax tree," in *Proc. IEEE 27th Int. Conf. Softw. Anal., Evol. Reengineering (SANER)*, Feb. 2020, pp. 261–271, doi: 10.1109/SANER48275.2020.9054857.

[39] Y. Wang, K. Wang, F. Gao, and L. Wang, "Learning semantic program embeddings with graph interval neural network," *Proc. ACM Program. Lang.*, vol. 4, pp. 1–27, Nov. 2020, doi: 10.1145/3428205.

[40] B. Wei, "Retrieve and refine: Exemplar-based neural comment generation," in *Proc. 34th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2019, pp. 1250–1252, doi: 10.1109/ASE.2019.00152.

[41] X.-C. Wen, Y. Chen, C. Gao, H. Zhang, J. M. Zhang, and Q. Liao, "Vulnerability detection with graph simplification and enhanced graph representation learning," 2023, *arXiv:2302.04675*.

[42] B. Wu and F. Zou, "Code vulnerability detection based on deep sequence and graph models: A survey," *Secur. Commun. Netw.*, vol. 2022, pp. 1–11, Oct. 2022, doi: 10.1155/2022/1176898.

[43] Y. Wu, J. Lu, Y. Zhang, and S. Jin, "Vulnerability detection in C/C++ source code with graph representation learning," in *Proc. IEEE 11th Annu. Comput. Commun. Workshop Conf. (CCWC)*, Jan. 2021, pp. 1519–1524, doi: 10.1109/CCWC51732.2021.9376145.

[44] Y. Wu, D. Zou, S. Dou, W. Yang, D. Xu, and H. Jin, "VulCNN: An image-inspired scalable vulnerability detection system," in *Proc. IEEE/ACM 44th Int. Conf. Softw. Eng. (ICSE)*, May 2022, pp. 2365–2376, doi: 10.1145/3510003.3510229.

[45] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *Proc. IEEE Symp. Secur. Privacy*, May 2014, pp. 590–604, doi: 10.1109/SP.2014.44.

[46] F. Yamaguchi, "Pattern-based vulnerability discovery," NiedersÃchsische Staats-und Universittsbibliothek GÃttingen, Göttingen, Germany, 2015.

[47] F. Yamaguchi, M. Lottmann, N. Schmidt, M. Pollmeier, S. Sharma, and C.-V. Ursache. *Code Property Graph Specification 1.1*. Accessed: Jun. 24, 2022. [Online]. Available: https://cpg.joern.io/

[48] Y. Yang, X. Xia, D. Lo, and J. Grundy, "A survey on deep learning for software engineering," *ACM Comput. Surv.*, vol. 54, no. 10s, pp. 1–73, Jan. 2022, doi: 10.1145/3505243.

[49] Z. Yu, W. Zheng, J. Wang, Q. Tang, S. Nie, and S. Wu, "CodeCMR: Cross-modal retrieval for function-level binary source code matching," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 33, 2020, pp. 3872–3883.

[50] X. Yuan, G. Lin, Y. Tai, and J. Zhang, "Deep neural embedding for software vulnerability discovery: Comparison and optimization," *Secur. Commun. Netw.*, vol. 2022, pp. 1–12, Jan. 2022, doi: 10.1155/2022/5203217.

[51] P. Zeng, G. Lin, L. Pan, Y. Tai, and J. Zhang, "Software vulnerability analysis and discovery using deep learning techniques: A survey," *IEEE Access*, vol. 8, pp. 197158–197172, 2020, doi: 10.1109/ACCESS.2020.3034766.

[52] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng. (ICSE)*, May 2019, pp. 783–794, doi: 10.1109/ICSE.2019.00086.

[53] J. Zhang, X. Wang, H. Zhang, H. Sun, and X. Liu, "Retrieval-based neural source code summarization," in *Proc. IEEE/ACM 42nd Int. Conf. Softw. Eng. (ICSE)*, Seoul, South Korea, Oct. 2020, pp. 1385–1397.

[54] K. Zhang, W. Wang, H. Zhang, G. Li, and Z. Jin, "Learning to represent programs with heterogeneous graphs," in *Proc. IEEE/ACM 30th Int. Conf. Program Comprehension (ICPC)*, May 2022, pp. 378–389, doi: 10.1145/3524610.3527905.

[55] Y.-Y. Zhang and M. Li, "Find me if you can: Deep software clone detection by exploiting the contest between the plagiarist and the detector," in *Proc. AAAI Conf. Artif. Intell.*, Jul. 2019, vol. 33, no. 1, p. 1, doi: 10.1609/aaai.v33i01.33015813.

[56] Y. Zhang and B. Li, "Malicious code detection based on code semantic features," *IEEE Access*, vol. 8, pp. 176728–176737, 2020, doi: 10.1109/ACCESS.2020.3026052.

[57] G. Zhao and J. Huang, "DeepSim: Deep learning code functional similarity," in *Proc. 26th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, Oct. 2018, pp. 141–151, doi: 10.1145/3236024.3236068.

[58] Y. Zheng, S. Pujar, B. Lewis, L. Buratti, E. Epstein, B. Yang, J. Laredo, A. Morari, and Z. Su, "D2A: A dataset built for AI-based vulnerability detection methods using differential analysis," 2021, *arXiv:2102.07995*.

[59] 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 *Proc. Adv. Neural Inf. Process. Syst.*, vol. 32, 2019, pp. 10197–10207.

[60] Y. Zhou, X. Yan, W. Yang, T. Chen, and Z. Huang, "Augmenting Java method comments generation with context information based on neural networks," *J. Syst. Softw.*, vol. 156, pp. 328–340, Oct. 2019, doi: 10.1016/j.jss.2019.07.087.

[61] Y. Zhuang, S. Suneja, V. Thost, G. Domeniconi, A. Morari, and J. Laredo, "Software vulnerability detection via deep learning over disaggregated code graph representation," 2021, *arXiv:2109.03341*.

[62] D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin, "μVulDeePecker: A deep learning-based system for multiclass vulnerability detection," *IEEE Trans. Dependable Secure Comput.*, vol. 18, no. 5, pp. 2224–2236, Sep. 2019, doi: 10.1109/TDSC.2019.2942930.

[63] *Software Assurance Reference Dataset*. Accessed: Mar. 12, 2022. [Online]. Available: https://samate.nist.gov/SARD/

[64] *NVD—Vulnerabilities*. Accessed: Jul. 27, 2023. [Online]. Available: https://nvd.nist.gov/vuln

JOSEPH GEAR received the bachelor's degree (Hons.) in information technology from the Queensland University of Technology, Brisbane, Australia, in 2020, where he is currently pursing the Ph.D. degree in computer science. His research interests include deep learning for vulnerability discovery and source code representation learning.

YUE XU (Member, IEEE) is currently an Associate Professor with the School of Computer Science, Queensland University of Technology, Australia. She is an active Researcher in the areas of data mining, machine learning, and web intelligence, and she has made important contributions to related research areas. She has published over 180 refereed articles, some of which have been published in top journals, such as IEEE Transactions on Knowledge and Data Engineering, *Information Science*, *European Journal of Information Systems*, *ACM Transactions on Intelligent Systems and Technology*, and *Knowledge-Based Systems*, and major conferences, such as ICDM, CIKM, WWW, and ICIS.

ERNEST FOO (Member, IEEE) has published over 100 refereed articles, including over 20 journal articles. His research can be broadly grouped into the field of applied cyber security. National critical infrastructure systems, such as electricity substations and water treatment plants are becoming more connected and are susceptible to cyber-attacks. His research interests include new ways to detect cyber attackers in critical infrastructure systems and prevent those attackers from being effective. The mechanisms he has employed in his research include machine learning, data mining, and cryptographic protocols and network simulations.

PRAVEEN GAURAVARAM received the Ph.D. degree in cryptology from the Queensland University of Technology, Brisbane, Australia.

He is currently a Consultant and a Senior Scientist with Tata Consultancy Services Ltd. (TCS), Australia and New Zealand (ANZ). He holds honorary academic titles of an Adjunct Associate Professor with UNSW and an Adjunct Professor with Deakin University. He is also an Advisor of the ICT Curriculum Advisory Group, Southern Cross University. He primarily manages and leads TCS's Co-Innovation (COIN) Research and Development Partnership with the Cyber Security Cooperative Research Centre (Cyber Security CRC) by actively engaging with the internal and external stakeholders to build, develop, and utilize emerging cyber security research. At TCS ANZ, he has also been influential in the business development, marketing, corporate affairs, and innovation branding activities. Simultaneously, he has committed to the national cyber security development activities and he has made significant contributions to the Australian Cyber Security Strategy, in 2020, National Quantum Strategy, in 2022, and other related Government initiatives, such as Security Legislation Amendment (Critical Infrastructure Protection) Act 2022. He has held scientific positions in India, Europe, and Australia. He was a recipient of research grants and awards whist his research fellowship with the Technical University of Denmark. He was a recipient of the Young Elite Researcher Award, in 2010, from the Danish Agency for Science, Technology, and Innovation for his contributions to research in Cryptographic Hash Functions. He has more than 60 publications and consulting advisories in cyber security.

ZAHRA JADIDI (Member, IEEE) received the Ph.D. degree in network security from the School of Information and Communication Technology, Griffith University. She is currently an Academic Member of Griffith University. She has published over 50 refereed articles. Her research interests include the security of cyber-physical systems, anomaly detection, and the automation of security analysis.

LEONIE SIMPSON is currently an Associate Professor and an Information Security Researcher with the School of Computer Science, Queensland University of Technology. She has been involved in information security research for over 25 years. She has authored or coauthored over 70 refereed research articles. Her research interests include symmetric cryptology and cyber security.