Software Vulnerability Detection via Multimodal Deep Learning

Xin Zhou(✉) and Rakesh M. Verma(✉)

University of Houston, Houston, TX, USA
xzhou21@uh.edu, rmverma2@central.uh.edu

Abstract. Vulnerabilities in software are like ticking time bombs, but it is difficult to completely eliminate them. For example, buffer overflow is a quite common vulnerability that occurs when a program receives too much data that can corrupt nearby space in memory and manipulate other data for malicious actions. To detect potential vulnerabilities in source code, we consider the code as multisource data by extracting semantically meaningful sub-graphs: Abstract Syntax Tree Graph (ASTG) and Tokenized Data Flow Graph (TDFG). We combine these with the original sequence of tokens and 49 heuristic features to train and leverage a multimodal deep learning network to detect vulnerable statements. We propose a Multisource Deep Learner (MDL) with joint representations based on the pretrained attention-based Bidirectional Gated Recurrent Unit (BGRU) neural networks for vulnerability detection in source code. Our framework not only detects potential vulnerabilities but also locates and ranks the vulnerable statements according to their importance based on the Program Dependence Graph (PDG). Our results show that an MDL-based model using multiple modalities is significantly better than a single modality based model. We also present comparisons with state-of-the-art methods.

Keywords: Static Analysis · Source Code · Software Bugs · Data Flow Graph · Abstract Syntax Tree · Deep Learning

1 Introduction

During the software development and deployment process, the later the bug is found, the greater the cost of repair. Most of the software defects are introduced in the coding stage, some of them escape detection in the current approaches of unit testing, integration testing, functional testing, and acceptance testing. InfoQ [39] reported that 30% to 70% of code logic design and coding defects can be discovered and repaired through static code analysis. Hicken et al. [14] also reported that, as expected, 85% of defects come in during the coding phase, but only a few defects are found during coding since we typically find bugs when we start testing the programs. Static code analysis plays a very critical role in the secure development process, and it must be moved forward as much as possible, since earlier detection can reduce the cost of development and repair for developers and companies. Many companies will likely encounter substantial resistance from developers to implement static code analysis tools due to the large number of false alarms that are generated. This means developers will waste considerable time in bug confirmation. Therefore, only a suitable and practicable static analysis tool can really reduce the development cost. There are two main static code analysis methods: 1) analyze intermediate files compiled from source code such as binary, language-independent intermediate representation (LLVM), etc., and 2) analyze source code directly through semantic information extracted from source files. Our framework is focused on source code itself.

According to [17,19,36], the main techniques for static code analysis are: 1) developing a defect pattern database and then matching the code to be analyzed with common defect patterns to detect potentially vulnerable statements. This method is simple and convenient but needs enough patterns and is typically prone to false positives. 2) Type inference refers to the automatic detection of the type of an expression in a formal language to ensure that each statement in the code has the correct type. 3) Model checking is based on finite state automata. The impact of each statement is abstracted into a state of a finite state automaton, and then the purpose of code analysis is achieved by analyzing the finite state machine. It can check timing characteristics such as program concurrency. 4) Data flow analysis by collecting semantic information from source code and abstracting it with a control flow graph. It can analyze and discover the behaviors of the program during run-time without actually running the program. 5) Data driven prediction using machine learning by utilizing the above analytical techniques based on a large training set that contains a diverse set of vulnerable and non-vulnerable patterns. We focus on data driven techniques.

Multi-modal learning involves relating information from multiple sources such as images and text. Multi-modal representation learning tries to eliminate redundancy and utilizes complementarity between modalities, so as to learn better features representation. Currently, there are two research directions in multi-modal learning: 1) joint representation, which refers to mapping the information of multiple modalities together into a unified multimodal vector space; 2) coordinated representations, which refers to mapping each modality to its respective representation space, but certain correlation constraints (such as linear correlation) are satisfied between the mapped vectors.

In computer vision, multi-modal learning has grown rapidly recently. Unstructured data can inherently take many forms such as visual and textual content. In this paper, we construct two type of modalities, i.e., sequential and graphical representations, from raw data using different constructors. Then, we focus on static vulnerability detection in source code via multi-modal learning and make the following contributions:

1. We propose a new tokenization method with abstract representation of numbers that outperforms state of art methods in rigorously repeated experiments with random train, valid, and test dataset splits and averaged results.
2. We create a multi-modal dataset for vulnerability detection in source code.

3. We propose the Multisource Deep Learner for vulnerability detection in source code via multi-modal learning.
4. We propose the Vulnerability Highlighter to locate vulnerable statements and rank the relevant statements.
5. We conduct a series of ablation experiments to show the value of significant components of our ML pipeline.

Organization. After the Related Work section, in Sect. 3, we explain how we extract and tokenize source code as four modalities from different perspectives. Section 4 details the data-driven prediction method that learns code patterns and dependency graph to detect the vulnerabilities and locate vulnerable statements. In Sect. 5, we describe the datasets used for the evaluation. Experimental details and results are discussed in Sect. 6 and Sect. 7 concludes the paper.

2 Related Work

We discuss the related work on this topic in four categories: custom token-based approaches, abstract syntax tree based approaches, data driven approaches, and multimodal learning based approaches.

Custom Token-Based Approaches: Russell et al. [33] design a function-level vulnerability detection system using machine learning. They compile millions of open-source functions and label them with carefully selected findings from three different static analyzers that indicate potential exploits. The authors have applied a variety of ML techniques inspired by classification problems in the natural language domain, fine-tuned them for their application, and achieved the best overall results via convolutional neural network and classified with an ensemble tree algorithm. However, function-level vulnerability detection is not as useful as statement-level detection in real-world detection, since functions can be too large (e.g., 4,000 and 12,000 line functions are mentioned in [25]) and time-consuming for an expert to manually investigate.

Abstract Syntax Tree-Based Approaches: Mark Weiser [40] designed a program slicing method for automatically decomposing programs by analyzing their data flow and control flow. The author mentions this program slicing method can be used for debugging and parallel processing of slices. Recently, several automatic vulnerability detection works are based on a similar idea of combining data flow, control flow, and Abstract Syntax Tree (AST).

VulDeePecker [24] is the first system showing the feasibility of using deep learning to detect vulnerabilities while being able to narrow down locations of vulnerabilities. The authors also present the first vulnerability dataset for deep learning approaches. VulDeePecker is only able to deal with vulnerabilities related to library/API function calls. Their newer framework SySeVR [23] is used to detect vulnerabilities in source code based on so-called Syntax-Semantics Vector Representation, which is extracted with known potential vulnerable characteristics related to function calls, array usage, pointer usage, or arithmetic expressions. They truncate or pad input as a set of fixed length sequences of tokens (threshold = 500) for neural networks.

The Vulnerability Deep learning-based Locator, VulDeeLocator [22], uses a deep learning-based fine-grained vulnerability detector for C source code. The authors detected four vulnerabilities that were not reported in the National Vulnerability Database (NVD), but their framework is limited to C programs and heavily relies on the LLVM compiler, since their representations are based on the LLVM intermediate representation.

Alon et al. designed a neural model, Code2Vec [2], for representating snippets of code as continuous distributed vectors. They demonstrate the effectiveness of their model to predict a method's name from the vector representation of its body based on the AST. However, their model is only able to predict labels that were observed as-is at training time and unable to compose such names and usually catches only the main idea. This paper inspired us to extend program text representation with different kinds of graph representations.

Other Data Driven Approaches with Different Features: Harer et al. [11] design a software vulnerability detection framework, which is a data-driven approach to detect vulnerabilities with machine learning in C and C++ programs. They use features based on the operations in each basic block (opcode, vector, or op-vecv) derived from a program build process using Clang and LLVM. Then, they combine this with source-based features using C/C++ lexer to predict vulnerability at the function level. Their work is limited by the labels of functions, since it is really hard to manually investigate and validate labels that are generated by other static analysis tools such as Clang static analyzer. Li et al. [21] present a vulnerability detector, based on sub-graphs in the Program dependence Graphs, that outputs the crucial statements that are relevant to the detected vulnerability.

Multimodal Learning Approaches: Heidbrink et al. [6,12,13] proposed a method that uses multimodal learning for flaw detection in software programs based on two modalities (source code and program binary). In source code, they extract subgraph information by counting all unique node-edge-node transitions and flaw analysis-inspired statistical features associated with following program constructs: function call (e.g., number of external calls), variables (e.g., number of explicitly defined variables), graph node counts (e.g., number of else statements), graph structure (degrees of AST nodes by type). For binaries, they used Ghidra to extract and collect statistical count information per function associated with function call, variables (e.g., number of stack variables), function size (e.g., number of basic blocks), and p-code opcode instances, which is Ghidra's intermediate representation language for assembly language instructions.

Other Approaches: In computer vision research improved model have been proposed based onmulti-view techniques. This line of research shows that analyzing an object from different perspectives can extract more semantic features and information. Jin et al. [15] proposed a method to take joint-embedding of shapes and contours. Lai et al. [20] introduced a large-scale, hierarchical multi-view object dataset RGB-D (Red-Green-Blue-Depth) collected using an RGB-D camera. RGB-D based object combines color and depth information to substantially improve results. Mokhov [26] designs a machine learning approach for static code analysis and fingerprinting for security bugs using the MARFCAT [27] application [10]. Sestili et al. [35] points towards future approaches that may solve vulnerability detection problems using representations of code that can capture appropriate scope information and using deep learning methods that are able to perform arithmetic operations. They developed a code generator to produce an arbitrarily large number of code samples of controlled complexity. They also investigated the limits of the current state-of-the-art AI system for detecting buffer overflows and compared it with current static analysis engines. Their data are simple C-like programs, which are generated as basic blocks without loops, conditionals, and variables with unknown value. Katz et al. [16] design a framework to convert a program in low-level representation back to a higher-level human-readable representation based on neural machine translation. Their framework can automatically learn a decompiler from a given compiler. However, their framework fails if the input is longer than the threshold value. Wang et al. [38] propose a graph neural network assisted data flow analysis method to find potential buffer overflows in execution traces. Yamaguchi et al. [41] employ the concept of code property graph in many graph databases such as ArangoDB, Neo4J, and OrientDB and demonstrate its efficacy by identifying 18 previously unknown vulnerabilities in the source code of the Linux Kernel.

3 Background and Approach

In this section, we first describe and explain how to extract and tokenize source code into different representations as different modalities. Second, we introduce and explain our framework for vulnerability detection in source code.

3.1 Data Representations

These four data representations are the modalities for multimodal learning.

1. Token: we extract and tokenize the sliced code into a sequence of lexical tokens based on the Program Dependence Graph (the definition is in Sect. 3.4).
2. Abstract Syntax Tree Graph (ASTG): is a graph type modality, which is generated by AST constructor.
3. Tokenized Data Flow Graph (TDFG): is a graph type modality, which is based on data flow dependencies.
4. Heuristic Features: the syntactic complexity properties of source code [4] (e.g., number of variable operations, number of function calls, etc.). Totally, we have 49 features [4] generated from the properties of AST and tokens.

ASTG and TDFG are extracted as structural semantic information similar to depth scans for images in computer vision. For example, when you consider a specific variable in source code, you focus only on the lines that use this variable.

3.2 Potential Vulnerable Statement

Potential Vulnerable Statement is a pre-defined collection from Li et al. [23] based on the Checkmarx over open-source tools Flawfinder [9] and RAT [31]. This collection is used for extracting program dependence graph and highlighting the vulnerabilities.

3.3 Abstract Syntax Tree

An AST is used to represent the abstract syntactic structure of source code in a formal language. Once we have the tree representation of source code, we can mine all possible paths through terminal-to-terminal, root-to-terminal, or other efficient kernels. We use an open-source tool ASTminer [18] to generate the ASTs. Then, we keep the same node ID for the same variables by merging all of them into one node to connect all edges for final AST Graph.

3.4 Program Dependence Graph

Program dependence graph (PDG) [8] consists of control dependency and data dependency, which are defined based on the Control Flow Graph (CFG).

Control Flow Graph (CFG)) [8]**:** For static analysis, the CFG is essential to extract semantic features and accurately represent the flow inside of a program unit. Let P be a program that consists of functions. The CFG of function f_i is a graph $G_i = (V_i, E_i)$, where V_i is a set of nodes, each node represents a statement or control predicate, and E_i is a set of directed edges such that each edge represents the possible flow of control between a pair of nodes.

Data Dependency [8]**:** Let P be a program that consists of functions and let the CFG for function f_i be $G_i = (V_i, E_i)$. A node n_{ik} will be considered as data dependent if there is a path from n_{ik} to n_{ij} in G_i and a value computed at node n_{ik} is used at node n_{ij}, where $1 \leqslant j, k \leqslant l_i$ and $j \neq k$, where l_i is total number of statements from f_i.

Control Dependency [8]**:** Let P be a program consisting of functions f_i with CFG $G_i = (V_i, E_i)$. If there exists a path starting at n_{ik} and ending at n_{ij} such that (i) n_{ij} post-dominates every node on the path excluding n_{ik} and n_{ij}, and (ii) does not post-dominate n_{ij}, then n_{ij} is control dependent on n_{ik}.

PDG [8]**:** Let P be a program that consisting of functions f_i with PDG $G'_i = (V_i, E'_i)$, where V_i is the same as G_i for CFG and E' is a set of directed edges such that each edge represents a data or control dependency between a pair of nodes.

3.5 TDFG and ASTG

Tokenized Data Flow Graph (TDFG) is constructed based on the tokenized program by the following steps: 1) collect potential vulnerable statement line numbers, 2) generate data flow graph based on these collected line numbers, 3) construct a graph G = (V, E) with tokenized source code where a node v_i represents a partial statement and an edge represents a the flow of data between a pair of nodes. Final TDFG feature set is a collection of sub-graphs from TDFG based on the potential vulnerable statements. Our previous work [42] shows how the TDFG is constructed and how sub-graphs are extracted. Abstract Syntax Tree Graph has the same extraction process as TDFG. Since AST tree can be directly represented as G = (V, E) where V is a set of nodes and E is a set of edges, where a node represents a token type and a edge represents a possible flow of control between a pair of nodes. We convert each potential vulnerable statement as a shared node (using same node index) over all modalities for alignment. Both TDFG and ASTG sequences and sub-graphs can be embedded as word or graph level embedding.

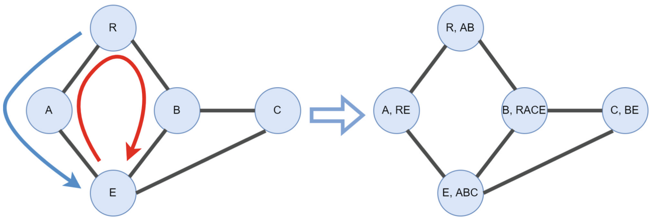
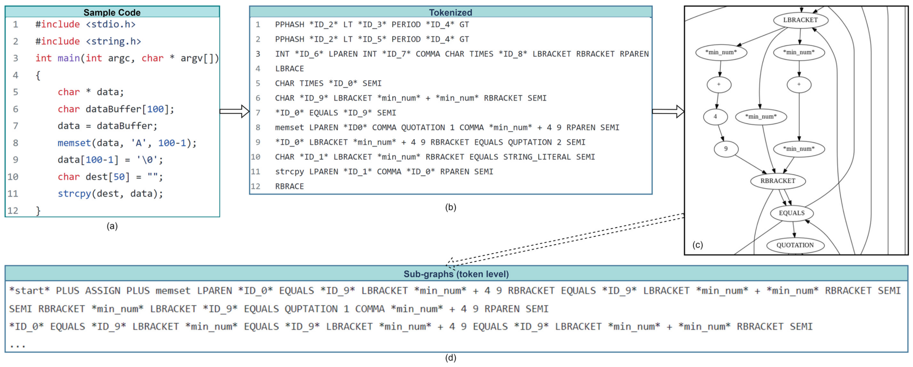


Fig. 1. A sub-graph sample. R is root node and E is exit node; Red arrow line is terminal to terminal path and blue arrow line is root to terminal path on the left; the right graph shows the first iteration of WLGK algorithm (Color figure online)

3.6 Sub-graph Extractions

We collect sub-graphs using the following three extraction methods to find semantic representations of source code:

1. Root-to-terminal (RTT): is a collection of paths from the root node to a terminal node.
2. Terminal-to-terminal (TTT): is a collection of paths from a terminal node to a terminal node. This method has been used by Code2Vec [2] and Code2Seq [1].
3. Weisfeiler-Lehman Graph Kernels (WLGK): [37] is a rapid feature extraction scheme based on the Weisfeiler-Lehman test of isomorphism on graphs. We use WLGK to walk through the paths and extract sub-graphs from both ASTG and TDFG since it has been found useful in other tasks, e.g., Graph2Vec [28].


Sample Code

```
#include <stdio.h>
#include <string.h>
int main(int argc, char * argv[])
{
    char * data;
    char dataBuffer[100];
    data = dataBuffer;
    memset(data, 'A', 100-1);
    data[100-1] = '\0';
    char dest[50] = "";
    strcpy(dest, data);
}
```

(a)

Tokenized

```
PPHASH *ID_2* LT *ID_3* PERIOD *ID_4* GT
PPHASH *ID_2* LT *ID_5* PERIOD *ID_4* GT
INT *ID_6* LPAREN INT *ID_7* COMMA CHAR TIMES *ID_8* LBRACKET RBRACKET RPAREN
LBRACE
CHAR TIMES *ID_0* SEMI
CHAR *ID_9* LBRACKET *min_num* + *min_num* RBRACKET SEMI
*ID_0* EQUALS *ID_9* SEMI
memset LPAREN *ID0* COMMA QUOTATION 1 COMMA *min_num* + 4 9 RPAREN SEMI
*ID_0* LBRACKET *min_num* + 4 9 RBRACKET EQUALS QUPTATION 2 SEMI
CHAR *ID_1* LBRACKET *min_num* RBRACKET EQUALS STRING_LITERAL SEMI
strcpy LPAREN *ID_1* COMMA *ID_0* RPAREN SEMI
RBRACE
```

(b)

Sub-graphs (token level)

```
*start* PLUS ASSIGN PLUS memset LPAREN *ID_0* EQUALS *ID_9* LBRACKET *min_num* + 4 9 RBRACKET EQUALS *ID_9* LBRACKET *min_num* + *min_num* RBRACKET SEMI
SEMI RBRACKET *min_num* LBRACKET *ID_9* EQUALS QUPTATION 1 COMMA *min_num* + 4 9 RPAREN SEMI
*ID_0* EQUALS *ID_9* LBRACKET *min_num* EQUALS *ID_9* LBRACKET *min_num* + 4 9 EQUALS *ID_9* LBRACKET *min_num* + *min_num* RBRACKET SEMI
...
```

(d)

Fig. 2. TDFG sub-graphs extraction example: (a) is a sliced sample for the model, (b) is tokenized PDG, (c) is TDFG, and (d) is a set of sub-graphs in token-level.

Weisfeiler-Lehman algorithm updates node attributes of a node v by:

$$h_i^{(t)}(v) = HASH(h_i^{(t-1)}(v), F\{h_i^{(t-1)}(u) \mid u \in N(v)\}) \tag{1}$$

where F is an aggregation function that concatenates topologically ordered neighbor's embedding, h_i is the ith attribute of v, u is v's neighbor node, and N is the set of neighbor nodes. The right part of Fig. 1 shows the first iteration of Weisfeiler-Lehman algorithm based on the left graph. For our sub-graph extraction, we collect the paths based on 1-dimensional Weisfeiler-Lehman algorithm with 5 iterations (after grid search from 1 to 10).

Representation: we extract and concatenate the sub-graphs as final representation (MAX = 500 tokens) using above methods based on the TDFG and ASTG. Figure 1 is an example of how sub-graphs are extracted by these three methods.

Figure 2(d) is an example of how a sequence of tokens is generated from raw sample code: line 1 is an example of an RTT path, line 2 is an example of TTT, and line 3 is an example of WLGK path in the sub-graphs.

3.7 Pipeline

We propose a multimodal learning framework for vulnerability detection in source code based on different modality extraction methods. Figure 3 shows the overview of our framework. We first generate Abstract Syntax Tree from source code and Program Dependence Graph from tokenized code. Then, we extract sub-graphs from AST as ASTG modality, tokens from PDG as Token modality and heuristic features (HF) from PDG as HF modality and extract sub-graphs from tokenized data flow graph as TDFG modality. The neural network could be any kind of multimodal leaning network to concatenate and align all modalities for final classification.

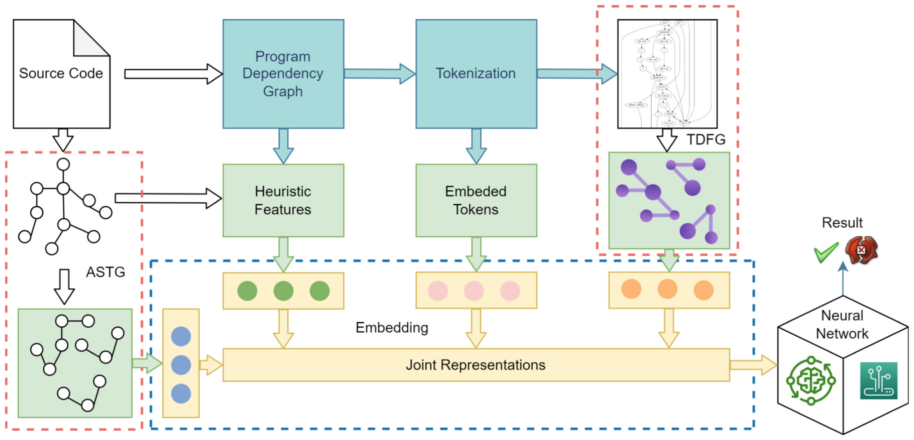
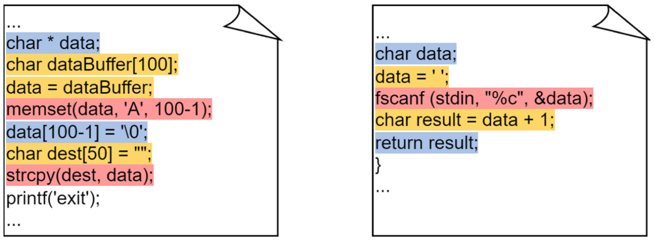


Fig. 3. Multisource Deep Learner Pipeline



```
...
char * data;
char dataBuffer[100];
data = dataBuffer;
memset(data, 'A', 100-1);
data[100-1] = '\0';
char dest[50] = "";
strcpy(dest, data);
printf('exit');
...
```

```
...
char data;
data = ' ';
fscanf (stdin, "%c", &data);
char result = data + 1;
return result;
}
...
```

Fig. 4. Vulnerability Highlighter is used to locate vulnerable statements; left example shows stack-based buffer overflow and the right example shows integer overflow.

3.8 Vulnerability Highlighter

We consider the pre-defined potential vulnerable tokens as the Most Possible Vulnerable Statements (MPVSs). If a program is detected as GOOD, we output the result without any highlights. If a program is detected as BAD, we proceed as below:

1. Denote all statements that contain MPVS label as M.
2. Generate control flow graph (CFG), data dependency, and control dependency to construct a program dependence graph (PDG) for each MPV.
3. Label all MPVSs ($[m_1, m_2, ..., m_n] \subseteq M$) with red (dangerous) background color in the program if it is detected as vulnerable.
4. For $i = 1$ to n, we extract their data and control dependencies for CFG G_i.
5. Union all forward slices as one forward list and all backward slices as one backward list respectively for data and control dependencies.
6. Label the statements with orange (warning) background color for backward data dependents of the MPVs if it is not in dangerous. Label other statements from PDG with blue (likely neutral) background color.

7. Keep all other statements, those that are not in PDG, with no highlights.

Figure 4 shows vulnerable statements found by the Vulnerability Highlighter.

4 Neural Network Models

We used convolutional neural network (CNN) for preliminary investigation on graph embedding and feature extraction methods, because of its speed for training and testing. Table 3 shows that Bidirectional Gated Recurrent Unit Neural Network (BGRU) [34] performs the best, in line with previous observations. Therefore, we use BGRU as the base model for further investigation.

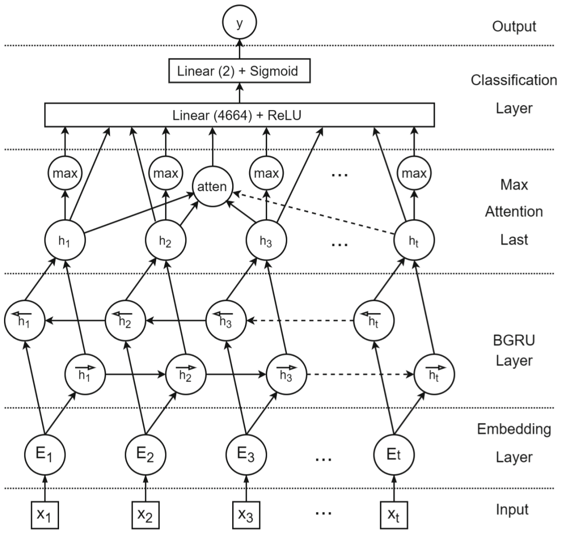


Fig. 5. Attention-based BGRU Classifier

4.1 Attention-Based BGRU

A Bidirectional GRU, or BGRU, is a sequence processing model that consists of two GRUs. One taking the input in a forward direction, and the other in a backwards direction. Gated recurrent units (GRUs) are a gating mechanism in recurrent neural networks, introduced by Kyunghyun Cho et al. [5]. Figure 5 shows how an attention-based BGRU classifier is constructed. Input can be either Token, ASTG, or TDFG. We use pretrained Word2Vec as embedding layer for each modality. A dot product attention layer is followed by BGRU layer. Then, we concatenate the output from attention layer, last hidden layer from BGRU, and max values of all elements from output of the last hidden layer of BGRU as a joint representation for final linear classifier with ReLU and Sigmoid activation functions. Our dot product attention layer is computed as follows:

$$a_t(s) = softmax((\frac{\exp(h_t^\intercal \overline{h}_s)}{\sum_{s'} \exp(h_t^\intercal \overline{h}_s)})$$

where a_t is output representation, s is input vector, and $\overline{h}_s$ is each source hidden state corresponding to the hidden target state h_t (Fig. 5).

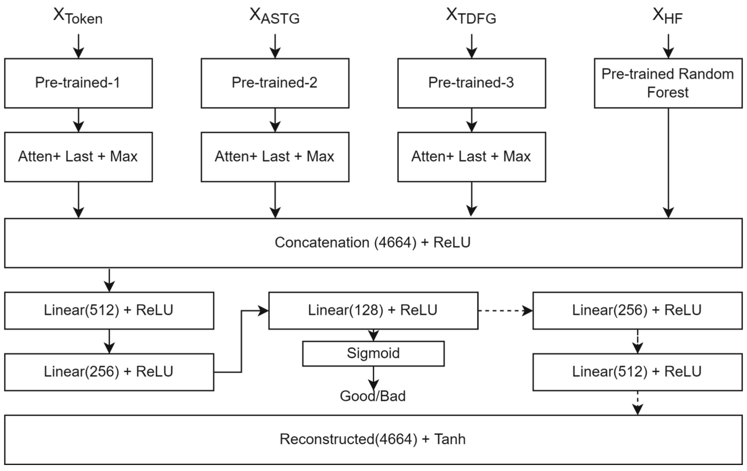


Fig. 6. Multisource Deep Learner

4.2 Multisource Deep Learner

We use three pretrained embedding layers and attention-based BGRU layer as the encoders for token, ASTG, and TDFG modalities. Then, we unfreeze pre-trained encoders (learned parameters can still be updated with 0.0001 learning rate) for correlational joint representations (size = 4,664) for vulnerability detection using our Multisource Deep Learner.

Multisource Deep Learner (MDL): it has a similar architecture as Correlational Neural Network (CorrNN [3]), but we use cross entropy loss function instead and added a classifier to fit our classification task. Our framework does not reconstruct all raw inputs, it reconstructs the joint representation by simple MLP encoder-decoder model to get semi-reconstruction loss to fine-tuning classification model using learning rate 0.0001. First, we take concatenated vector [x1, x2, x3, x4] of size d1 + d2 + d3 + d4 from the pooled layer (Attention + Last + Max) based on three pre-trained BGRU and one pre-trained random forest. Given z = (x1, x2, x3, x4), the first hidden layer computes an encoded representation as

$$h_1(z) = f(w_1 x_1 + w_2 x_2 + w_3 x_3 + w_4 x_4 + b) \tag{2}$$

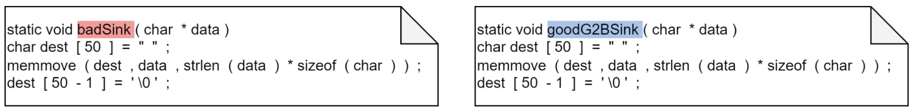
where w is a projection matrix and b is bias vector. Function f can be any non-linear activation function. We grid searched to find the best activation function ReLU for our framework. Our latent vector h is used for classification. We use Binary Cross Entropy (BCE) loss for training. BCE is computed as follows:

$$BCE = -(y \log(p) + (1 - y) \log(1 - p)) \tag{3}$$

where log is the natural log, y is binary indicator and p is predicted probability. We also tested with combined loss by summing up BCE of the classification and Mean Square Error (MSE) loss of the concatenation reconstruction.

5 Dataset

We use the MVDSC dataset [42], which is generated based on two sources: NVD [30] and SARD [29]. SySeVR dataset is also extracted from the same raw datasets, but it contains more than 10,000 mislabeled instances (e.g., see Fig. 7) and duplicates. However, we still use SySeVR [23] dataset as a baseline to compare with our single-modality based model, which investigates different tokenization methods for source code. MVDSC is a dataset generated without any duplicates. MVDSC-Mixed is a combination of MVDSC dataset and a small portion of synthetic instances. All these datasets are focused on the vulnerabilities that can be learned from vulnerable and non-vulnerable patterns such as buffer-related (overflow, underflow, etc.), integer-related (overflow, underflow, etc.), divide-by-zero, double-free, etc. For more details, please refer to NVD [30], SARD [29], and MVDSC [42].



```
static void badSink ( char  * data )
char dest  [ 50  ]  =  " "  ;
memmove  ( dest  , data  , strlen  ( data  )  * sizeof  ( char  )  )  ;
dest  [ 50  - 1  ]  =  ' \0 '  ;
```

```
static void goodG2BSink ( char  * data )
char dest  [ 50  ]  =  " "  ;
memmove  ( dest  , data  , strlen  ( data  )  * sizeof  ( char  )  )  ;
dest  [ 50  - 1  ]  =  ' \0 '  ;
```

Fig. 7. Two code snippets from SySeVR dataset that are identical except for function names, but they label the left as vulnerable, and the right one as non-vulnerable.

5.1 Preprocessing and Tokenization

Each program consists of one or more functions in NVD [30] and SARD [29]. Each function contains labels and comments about vulnerability details including how to fix. Therefore, we need to mask or remove sensitive information that may benefit models. We convert all file names and any token, that contains "bad", 'good', or 'cwe' sub-string (cwe_* contains sensitive information about vulnerabilities), to a fixed common string *C* with star symbols around to avoid code conflicts. We also convert all strings with single quotation mark as '*SQ* + n' and double quotation mark as '*DQ* + n' where n is the length of content in quotation. In addition, we remove all comments. We are using 811 pre-defined

Table 1. Dataset statistics (vulnerable: non-vulnerable).

Dataset	**train**	**valid**	**test**
SySeVR	pool: 64403 (13603:50800)		
MVDSC	7569:22416	1914:5580	1857:5637
MVDSC-Mixed	11416:26569	2401:6093	2325:6169

vulnerable syntax characteristics (memset, strcpy, etc.) which is generated by Li et al. [23] since we use the same raw dataset. We use pycparser [7] as our base lexer to find identifiers including variables and functions (finding identifiers can be tricky). We convert all variable and function names into more semantic meaningful representations (Table 1).

Locate_ID: for masking variable and function names, we need to index them. To keep the index order meaningful, we always index destination (sink) variable before source variable. Ex. strcpy(dest, src) will always be masked as strcpy(*ID_0*, *ID_1*), no matter which variable was declared first. To align those variables which are related to potential vulnerable statement, we denote the variables which are the closest to a potential vulnerable statement starting from 0. That means we can ensure that '*ID_0*' and '*ID_1*' are the two aligned tokens to vulnerable statements since most of function calls take two arguments in our dataset. A more complex function with more arguments can also be handled.

Abstract: after 'Locate_ID', we tokenize the remaining program units based on their types. Once the whole PDG is tokenized, we apply a number abstraction function, **Abstract()**, to convert numbers as (*MIN*, difference) in data flow statements only based on the potential vulnerable statement, where *MIN* represents the minimum number value of all numbers in these data dependents.

6 Experimental Results and Analysis

We now present the results of our experiments and ablation studies.

Metrics. We use accuracy (A), precision (P), recall (R), and F1 as our evaluation metrics. Our dataset is highly skewed since vulnerable statements are far fewer than non-vulnerable statements, so we add extra metric Matthews Correlation Coefficient (MCC) for evaluation.

Comparing Single-Modality Model with Baseline: we use SySeVR [23] as our baseline for single-modality model, since it was developed for detection originally from the same sources as MVDSC dataset [42]. SySeVR dataset contains 64,403 instances and the authors reported their results based on randomly picked dataset 30000/7500/7500 as train/valid/test, we also randomly picked with random seed from the pool with same split ratio. *We report average and*

Table 2. Tokenization comparison using SySeVR dataset; 10 run in 10 different random seeds. SySeVR-BGRU [23] was the best previous result. T is our tokenization method.

Method	A	P	R	F1	MCC
SySeVR-BGRU	94.7	91.5	n/a	86.8	83.6
T + CNN	94.7±0.4	87.6±2.1	87.5±1.7	87.5±0.9	84.2±1.1
T + BGRU	95.3±0.2	90.5±1.7	87.2±2.4	88.8±0.8	85.9±0.9

standard deviation in 10 runs with 10 random seeds, since it is a better evaluation method [32]. Table 2 shows that our single-modality model with same BGRU model as theirs is significantly better than their best result. Hence, we only use the MVDSC dataset [42] for further investigation. For the following experiments, we report the average of three runs in the same train/valid/test sets.

Table 3. Comparing models with token modality on MVDSC dataset [42]

Network	A	P	R	F1	MCC
CNN	95.5	90.4	91.4	90.9	87.9
LSTM	91.9	85.5	81.1	83.3	78.0
BLSTM	95.8	92.1	90.7	91.4	88.6
GRU	96.1	94.9	89.2	91.9	89.5
BGRU	96.6	94.3	91.6	93.0	90.7

Comparing token modality with different models: to find the best model for single-modality and build some pretrained models, we evaluated our token modality with five common networks on the MVDSC dataset. Table 3 shows that the Bidirectional-GRU classifier achieved the best performance among CNN, LSTM, Bidirectional-LSTM, and GRU. The table also shows that both bidirectional LSTM and GRU are better than LSTM and GRU respectively. This suggests that both backward and forward paths are useful for vulnerability detection.

Table 4. Comparing graph embedding in TDFG and ASTG

TDFG2Vec	A	P	R	F1	MCC
token	90.0	82.1	76.6	79.2	72.8
graph	84.7	76.7	55.0	64.1	55.9
ASTG2Vec	Acc	Pre	Recall	F1	MCC
token	91.5	83.7	81.5	82.6	77.0
graph	84.8	72.1	63.0	67.2	57.6

Comparing embedding methods for graph modality: we compare sub-graph embedding and Graph2Vec [28]. For sub-graph embedding, we concatenate all extracted paths as a long sequence (MAX = 500 tokens) and then use a Word2Vec embedding + BGRU (Fig. 5) that is connected with a dot product attention layer for classification. For graph embedding, we use a standard Graph2Vec [28] to embed ASTG or TDFG into a 1024-dimension vector with 5 Weisfeiler-Lehman iterations, then normalize it as a 32×32 grey scale image with a standard CNN classifier. Table 4 shows that token level embedding method is significantly better than graph level embedding. So, we embed a set of sub-graphs as 500×32 matrix for further experiments.

Table 5. Comparing tokenization methods on MVDSC dataset

Normal	Locate ID	Abstract	A	P	R	F1	MCC
✓			96.1	94.0	89.8	92.0	89.5
✓	✓		96.3	94.6	89.8	92.4	90.0
✓	✓	✓	96.6	94.3	91.6	93.0	90.7

Comparing tokenization with add-ons: Table 5 shows the differences between different tokenization methods. Two add-ons (Locate_ID and Abstract) eventually and slightly improved the model. With the abstract representation of numbers, the recall is increased by 1.8 which is a critical improvement in vulnerability detection since the size is very sensitive in memory allocation such as malloc()→free().

Table 6. Freezing vs Unfreezing the parameters of pre-trained models

Method	A	P	R	F1	MCC
4 modalities + BCE + freeze	97.0	96.3	91.3	94.8	91.8
4 modalities + BCE + unfreeze	97.7	97.2	93.4	95.2	93.8
4 modalities + CombinedLoss + freeze	95.1	98.5	81.5	89.2	86.7
4 modalities + CombinedLoss + unfreeze	97.8	97.0	93.9	95.4	94.0

Comparing freeze/unfreeze: we compared multiple modalities with frozen and unfrozen mode and tested with two loss functions. Table 6 shows that both unfrozen encoders worked significantly better than their frozen ones. CombinedLoss is not significantly different from BCE but made model training much slower. Therefore, we use BCE for further comparisons. We can see that unfreezing the parameters of the pre-trained model is a better way for fine-tuning.

Table 7. Ablation study of modalities on MVDSC dataset

Modalities	A	P	R	F1	MCC
Token	96.6	94.3	91.6	93.0	90.7
Token + ASTG	97.1	92.3	96.1	94.2	92.2
Token + ASTG + TDFG	97.6	96.9	92.9	94.8	93.2
Token + ASTG + TDFG + HF	97.7	97.2	93.4	95.2	93.8

Comparing single modality and multiple modalities using MVDSC dataset: this ablation study is used to learn how modalities can be stacked up and improve the classification performance in MVDSC dataset. Table 7 shows that all combined model has the best overall performance. The result also shows that ASTG is the booster for higher recall. TDFG and HF make the model more balanced for precision. Comparing token-modality to four combined modalities, the MCC is increased by 3.1% which is significantly better.

Table 8. Model comparisons using MVDSC-Mixed dataset

Modalities	A	P	R	F1	MCC
Token	94.2	94.8	83.6	88.8	85.3
Token + ASTG	95.2	91.0	92.0	91.3	88.0
Token + ASTG + TDFG	95.6	94.1	89.6	91.8	88.9
Token + ASTG + TDFG + HF	95.5	92.7	90.8	91.7	88.7

Table 9. MVDSC vs MVDSC-Mixed

Representations	MVDSC		MVDSC-Mixed		Differences	
	R	MCC	R	MCC	ΔR	ΔMCC
Token	91.6	90.7	83.6	85.3	-8.0	-5.4
Token + ASTG	96.1	92.2	92.0	88.0	-4.1	-4.2
Token + ASTG + TDFG	92.9	93.2	89.6	88.9	-3.3	-4.3
Token + ASTG + TDFG + HF	93.4	93.8	90.8	88.7	-2.6	-5.1

Comparing single modality and multiple modalities using MVDSC-mixed dataset: MVDSC-Mixed adds around 10% adversarial data to MVDSC. Table 8 shows that all modalities are negatively impacted by adversarial data. Table 9 shows that single-modality based model is the most negatively impacted to both recall (−8.0%) and MCC (−5.4%). Therefore, using multiple modalities not only improves the detection performance but also improves the robustness of the model.

7 Conclusion

We propose Multisource Deep Learner, a multimodal learning framework to detect vulnerabilities in source code and show their location in code. The framework mines semantic information for developers. We compared our framework with state-of-the-art algorithms from previous works. We evaluated our system with our multi-modal dataset MVDSC [42]. Our results show that multi-modality-based models are significantly better in performance and robustness than single-modality-based models by the dataset-based evaluation.

Acknowledgments. Research partially supported by NSF grants 1433817 and 2210198, ARO grant W911NF-20-1- 0254, and ONR award N00014-19-S-F009. Verma is the founder of Everest Cyber Security and Analytics, Inc.

A Appendix

A.1 Limitations

Apart from the usual limitations of static analysis and machine learning, other limitations are: 1) adversarial data may negatively impact model's performance, 2) the current implementation does not address interprocedural analysis.

References

1. Alon, U., Brody, S., Levy, O., Yahav, E.: Code2seq: generating sequences from structured representations of code. In: International Conference on Learning Representations (2019). https://openreview.net/forum?id=H1gKYo09tX
2. Alon, U., Zilberstein, M., Levy, O., Yahav, E.: Code2vec: learning distributed representations of code. Proc. ACM Program. Lang. **3**(POPL) (2019). https://doi.org/10.1145/3290353
3. Chandar, S., Khapra, M.M., Larochelle, H., Ravindran, B.: Correlational neural networks. Neural Comput. **28**(2), 257–285 (2016). https://doi.org/10.1162/NECO_a_00801
4. Chernis, B., Verma, R.: Machine learning methods for software vulnerability detection. In: Proceedings of the Fourth ACM International Workshop on Security and Privacy Analytics, pp. 31–39 (2018)
5. Chung, J., Gulcehre, C., Cho, K., Bengio, Y.: Empirical evaluation of gated recurrent neural networks on sequence modeling. In: NIPS 2014 Workshop on Deep Learning, December 2014 (2014)
6. Cooper, A., Zhou, X., Heidbrink, S., Dunlavy, D.M.: Using neural architecture search for improving software flaw detection in multimodal deep learning models. arXiv:2009.10644 (2020)
7. Eliben: Complete c99 parser in pure python: pycparser v2.21. https://github.com/eliben/pycparser/blob/master/pycparser. Accessed Nov 2021
8. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. (TOPLAS) **9**(3), 319–349 (1987). https://doi.org/10.1145/24039.24041

9. Flawfinder: Flawfinder. https://dwheeler.com/flawfinder/. Accessed Feb 2022
10. SQ Group: Static analysis tool exposition (SATE) VI workshop. https://www.nist.gov/itl/ssd/software-quality-group/static-analysis-tool-exposition-sate-vi-workshop. Accessed Mar 2022
11. Harer, J.A., et al.: Automated software vulnerability detection with machine learning. arXiv abs/1803.04497 (2018)
12. Heidbrink, S., Rodhouse, K.N., Dunlavy, D.M.: Multimodal deep learning for flaw detection in software programs. arXiv:2009.04549 (2020)
13. Heidbrink, S., Rodhouse, K.N., Dunlavy, D., Cooper, A., Zhou, X.: Joint analysis of program data representations using machine learning for improved software assurance and development capabilities (2020). https://doi.org/10.2172/1670527. https://www.osti.gov/biblio/1670527
14. Hicken, A.: The shift-left approach to software testing. https://www.stickyminds.com/article/shift-left-approach-software-testing. Accessed Mar 2022
15. Jin, A., Fu, Q., Deng, Z.: Contour-based 3D modeling through joint embedding of shapes and contours. In: Symposium on Interactive 3D Graphics and Games, I3D 2020. Association for Computing Machinery, New York (2020). https://doi.org/10.1145/3384382.3384518
16. Katz, O., Olshaker, Y., Goldberg, Y., Yahav, E.: Towards neural decompilation. arXiv abs/1905.08325 (2019)
17. Kotenko, I., Izrailov, K., Buinevich, M.: Static analysis of information systems for IoT cyber security: a survey of machine learning approaches. Sensors **22**(4) (2022). https://doi.org/10.3390/s22041335. https://www.mdpi.com/1424-8220/22/4/1335
18. Kovalenko, V., Bogomolov, E., Bryksin, T., Bacchelli, A.: PathMiner: a library for mining of path-based representations of code. In: Proceedings of the 16th International Conference on Mining Software Repositories, pp. 13–17. IEEE Press (2019)
19. Kulenovic, M., Donko, D.: A survey of static code analysis methods for security vulnerabilities detection. In: 2014 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), pp. 1381–1386 (2014). https://doi.org/10.1109/MIPRO.2014.6859783
20. Lai, K., Bo, L., Ren, X., Fox, D.: A large-scale hierarchical multi-view RGB-D object dataset. In: 2011 IEEE International Conference on Robotics and Automation, pp. 1817–1824 (2011). https://doi.org/10.1109/ICRA.2011.5980382
21. Li, Y., Wang, S., Nguyen, T.N.: Vulnerability detection with fine-grained interpretations, pp. 292–303. Association for Computing Machinery, New York (2021). https://doi.org/10.1145/3468264.3468597
22. Li, Z., Zou, D., Xu, S., Chen, Z., Zhu, Y., Jin, H.: VulDeeLocator: a deep learning-based fine-grained vulnerability detector. IEEE Trans. Dependable Secure Comput. **19**(4), 2821–2837 (2022). https://doi.org/10.1109/TDSC.2021.3076142
23. Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y., Chen, Z.: SySeVR: a framework for using deep learning to detect software vulnerabilities. IEEE Trans. Dependable Secure Comput. 1 (2021). https://doi.org/10.1109/tdsc.2021.3051525
24. Li, Z., et al.: VulDeePecker: a deep learning-based system for vulnerability detection. In: 25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, 18–21 February 2018. The Internet Society (2018). http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_03A-2_Li_paper.pdf
25. McConnell, S.: Code Complete. Pearson Education (2004)

26. Mokhov, S.A.: The use of machine learning with signal- and NLP processing of source code to fingerprint, detect, and classify vulnerabilities and weaknesses with MARFCAT. arXiv, Cryptography and Security (2011)
27. Mokhov, S.A., Paquet, J., Debbabi, M.: MARFCAT: fast code analysis for defects and vulnerabilities. In: 2015 IEEE 1st International Workshop on Software Analytics (SWAN), pp. 35–38 (2015). https://doi.org/10.1109/SWAN.2015.7070488
28. Narayanan, A., Chandramohan, M., Venkatesan, R., Chen, L., Liu, Y., Jaiswal, S.: Graph2vec: learning distributed representations of graphs. arXiv abs/1707.05005 (2017)
29. NIST: Software assurance reference dataset. https://samate.nist.gov/SRD/index.php. Accessed Mar 2022
30. NIST: National vulnerability database. https://nvd.nist.gov/. Accessed Nov 2021
31. RAT: rough-auditing-tool-for-security. https://code.google.com/archive/p/rough-auditing-tool-for-security/. Accessed May 2022
32. Reimers, N., Gurevych, I.: Reporting score distributions makes a difference: performance study of LSTM-networks for sequence tagging. In: Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing, Copenhagen, Denmark, pp. 338–348. Association for Computational Linguistics (2017). https://doi.org/10.18653/v1/D17-1035. https://aclanthology.org/D17-1035
33. Russell, R., et al.: Automated vulnerability detection in source code using deep representation learning. In: 2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA), pp. 757–762 (2018). https://doi.org/10.1109/ICMLA.2018.00120
34. Schuster, M., Paliwal, K.: Bidirectional recurrent neural networks. IEEE Trans. Signal Process. **45**(11), 2673–2681 (1997). https://doi.org/10.1109/78.650093
35. Sestili, C.D., Snavely, W., VanHoudnos, N.M.: Towards security defect prediction with AI. arXiv abs/1808.09897 (2018)
36. Sharma, T., Kechagia, M., Georgiou, S., Tiwari, R., Sarro, F.: A survey on machine learning techniques for source code analysis. arXiv abs/2110.09610 (2021)
37. Shervashidze, N., Schweitzer, P., van Leeuwen, E.J., Mehlhorn, K., Borgwardt, K.M.: Weisfeiler-lehman graph kernels. J. Mach. Learn. Res. **12**(77), 2539–2561 (2011). http://jmlr.org/papers/v12/shervashidze11a.html
38. Wang, Z., Yu, L., Wang, S., Liu, P.: Spotting silent buffer overflows in execution trace through graph neural network assisted data flow analysis. arXiv (2021). https://arxiv.org/abs/2102.10452
39. Wanjia: This 66-year-old is still writing code and wants to fix bugs early in the SDLC. https://xcalibyte.com/. Accessed Mar 2022
40. Weiser, M.: Program slicing. IEEE Trans. Softw. Eng. **SE-10**(4), 352–357 (1984). https://doi.org/10.1109/TSE.1984.5010248
41. Yamaguchi, F., Golde, N., Arp, D., Rieck, K.: Modeling and discovering vulnerabilities with code property graphs. In: 2014 IEEE Symposium on Security and Privacy, pp. 590–604 (2014). https://doi.org/10.1109/SP.2014.44
42. Zhou, X., Verma, R.M.: Vulnerability detection via multimodal learning: datasets and analysis. In: ASIA Conference on Computer and Communications Security (2022). https://doi.org/10.1145/3488932.3527288