



VulSim: Leveraging Similarity of Multi-Dimensional Neighbor Embeddings for Vulnerability Detection

Samiha Shimmi, Ashiqur Rahman, and Mohan Gadde, *Northern Illinois University*;
Hamed Okhravi, *MIT Lincoln Laboratory*; Mona Rahimi, *Northern Illinois University*

<https://www.usenix.org/conference/usenixsecurity24/presentation/shimmi>

**This paper is included in the Proceedings of the
33rd USENIX Security Symposium.**

August 14-16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

**Open access to the Proceedings of the
33rd USENIX Security Symposium
is sponsored by USENIX.**

VulSim: Leveraging Similarity of Multi-Dimensional Neighbor Embeddings for Vulnerability Detection

Samiha Shimmi
Northern Illinois University
sshimmi@niu.edu

Ashiqur Rahman
Northern Illinois University
ashiqur.r@niu.edu

Mohan Gadde
Northern Illinois University
mgadde1@niu.edu

Hamed Okhravi
MIT Lincoln Laboratory
hamed.okhravi@ll.mit.edu

Mona Rahimi
Northern Illinois University
mrahimi1@niu.edu

Abstract

Despite decades of research in vulnerability detection, vulnerabilities in source code remain a growing problem, and more effective techniques are needed in this domain. To enhance software vulnerability detection, in this paper, we first show that various vulnerability classes in the C programming language share common characteristics, encompassing semantic, contextual, and syntactic properties. We then leverage this knowledge to enhance the learning process of Deep Learning (DL) models for vulnerability detection when only sparse data is available. To achieve this, we extract multiple dimensions of information from the available, albeit limited, data. We then consolidate this information into a unified space, allowing for the identification of similarities among vulnerabilities through nearest-neighbor embeddings. The combination of these steps allows us to improve the effectiveness and efficiency of vulnerability detection using DL models. Evaluation results demonstrate that our approach surpasses existing State-of-the-art (SOTA) models and exhibits strong performance on unseen data, thereby enhancing generalizability.

1 Introduction

Software vulnerabilities continue to be a major source of financial and reputational harm to corporations [52, 102]. Despite intensive efforts from academia and industry to mitigate software vulnerabilities, the number of reported vulnerabilities in the Common Vulnerability and Exposure (CVE) database has increased over time [62, 70]. For instance, in 1999, a mere 321

CVE records were reported, whereas in 2023, this figure has skyrocketed to around 29K. Effective detection of software vulnerabilities is still a major and growing need.

One possible approach for detecting software vulnerabilities is the usage of DL models. While DL models have been used successfully in various other contexts, their adoption for vulnerability detection faces multiple major challenges.

The lack of large-scale, publicly available, and reliable labeled datasets is extensively discussed in the literature as a major challenge associated with the application of DL models in vulnerability detection [40, 114]. Note that databases such as the National Vulnerability Database (NVD) and SARD [81] do exist, but the amount of data they contain is not sufficient for training DL models. First, given the vast complexity of modern software and its large number of degrees of freedom, perhaps many millions of training samples would be necessary to train DL models, while the entirety of the NVD, spanning many decades and all types of software and vulnerability types, only contains around 220K vulnerabilities at the time of writing this paper. It is worth mentioning that, this figure represents reported vulnerabilities, requiring further processing by retrieving corresponding information from GitHub links, if available. However, the actual number of *usable vulnerable samples after processing is considerably lower*, as evidenced by studies [95, 116].

Moreover, the widely used SARD [81] dataset contains *synthetic samples*, and as Chakrabarty et al. [13] demonstrated, real-world examples are more complex than the synthetic counterparts. The performance drop was observed to be 54% in cases where the model was trained exclusively on non-synthesized data [13, 121]. Additionally, recent work by Chen et al. [16] demonstrates that increasing the volume of the training data does not necessarily enhance the performance of the model and can reach a saturation point. This was observed when they applied their recently compiled large DiverseVul dataset to several models.

Another limitation of the current datasets is data *imbalance* where the number of vulnerable samples is substantially lower than non-vulnerable ones. Models trained on

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

This material is based upon work supported by the Under Secretary of Defense for Research and Engineering under Air Force Contract No. FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Under Secretary of Defense for Research and Engineering.

imbalanced datasets are biased toward non-vulnerable samples [13]. Based on this finding, in order to get a good performance, one has to include approximately an equal number of non-vulnerable samples when training DL models. This implies that we cannot utilize all the samples from the existing datasets.

To summarize, existing vulnerability datasets have a relatively small number of usable samples necessary for proper training of DL models, many existing samples are synthetic, and there is an imbalance between the vulnerable and non-vulnerable samples in the datasets. These reasons contribute to general *poor performance* of DL models for vulnerability detection, and even *worse generalizability* (e.g., F1 score in SOTA dropping from 49% to 9.4%) [13, 16, 121]. With these challenges in mind, we hypothesize and later validate the effectiveness of leveraging more properties from the existing limited data to develop a more robust vulnerability detection framework.

To address the above-mentioned challenges, we observe that vulnerabilities often share common characteristics across multiple primary dimensions, including meanings of the tokens (semantic), implementation purpose (context), and code structure (syntax). We aim to leverage these properties along with their neighboring information to design a vulnerability prediction model with available, limited amounts of data. Throughout the rest of the paper, when we use the term neighbor, it refers to neighborhood using the Cosine similarity metric for each code embedding.

In this work, we introduce VulSim, a technique for **Vulnerability** detection based on multi-dimensional **Similar** neighbors. Initially, we conduct an in-depth analysis of multiple codebases and subsequently leverage this knowledge in VulSim. We consolidate the multi-dimensional information into a unified space and assisted in better vulnerability detection by leveraging insights from nearest neighbors. It is worth mentioning that while we leveraged three existing models SBERT [77], Code2vec [6], and CodeBERT [28] to capture semantic, contextual, and syntactic properties respectively, the novelty of our approach lies in consolidating all three dimensions alongside neighboring information. This approach enables the development of a robust vulnerability detection model that surpasses the performance of each individual model mentioned, including other SOTA approaches.

Our work highlights the discriminative capability of multiple dimensions in code vulnerability analysis and introduces a new framework that effectively captures valuable insights from diverse code embeddings. By considering multiple dimensions from similar data, our approach maximizes the available information and mitigates the dataset limitation problem. Along with retrieving information from different dimensions, we also take the neighboring information into account. To verify the generalizability of our approach, we train our model on one dataset and test it on a completely new dataset with unseen code samples.

In summary, we address the following research questions:

- RQ_1 : Are there dimensions, along which code vulnerabilities are more commonly similar?
- RQ_2 : How does the guided training approach improve a) the precision and recall rates and b) the generalizability of DL models for vulnerability detection in comparison to SOTA methods?

To overcome all the challenges we discussed earlier, our work makes the following contributions:

- **Addressing dataset limitations by consolidating multiple properties and neighboring information:** To compensate for exiguous data, we leverage the commonalities spanning over multiple vulnerability dimensions. Rather than expanding the size of a dataset, we focus on enhancing the depth of information derived from a limited number of samples. Unlike existing approaches that focus on homogeneous properties, such as source code or natural language vulnerability definitions [80, 98, 118, 120], our approach tackles the challenge of consolidating information from diverse properties. This approach enables us to draw conclusions, even in situations where additional data might have been helpful. Additionally, we consider the neighbor information while detecting if a specific code is vulnerable or not. This approach allows us to retrieve more information from the limited amount of available data.
- **Enhancing generalizability:** We illustrate that by capturing code vulnerabilities in multiple dimensions and focusing the DL training process on primary dimensions, we achieve improved generalizability of the trained models and enable them to effectively detect new or unseen vulnerabilities in real-world scenarios.

The evaluation results demonstrate that VulSim achieved an accuracy of 75%, surpassing the SOTA techniques on Microsoft CodexGLUE benchmark [56], where the leading model attained an accuracy of 69.29%. Moreover, our model demonstrated its ability to generalize to an entirely new dataset, achieving an accuracy of approximately 55% and a recall of 85%. Our experimental results also demonstrate that, in each dimension, the neighbor-based model was able to uniquely identify several instances of vulnerabilities which other dimensions failed to capture and thus demonstrate the power of merging all three dimensions. These findings highlight the robustness and effectiveness of VulSim in vulnerability detection tasks, even when applied to diverse and previously unseen code samples. The artifacts of this paper are publicly available online at <https://github.com/SamihaShimmi/VulSim/tree/main>.

Section 2 describes the necessary background information and Section 3 describes VulSim's implementation. Section 4 presents the evaluation results of the proposed approach.

Section 5 discusses VulSim's limitations. We discuss the related work in Section 6 and conclude the paper in Section 7.

2 Code Similarity Dimensions and Initial Observations

Our technique relies on analyzing code similarity in multiple dimensions: contextual, semantic, and syntactic. In this section, we first provide a definition of each dimension and provide examples to illustrate how different code segments can be similar in one of these dimensions. In the following subsections, we describe how we generate the embeddings from functions by leveraging some existing models to get different sets of properties. In the subsequent sections, we leverage this knowledge to build VulSim.

Throughout the rest of the paper, we commonly refer to the Common Vulnerability and Exposure (CVE) [62] and Common Weakness Enumeration (CWE) [63]. While the former is used to identify a specific vulnerability in a specific piece of code (e.g., a buffer overflow at function x of source file y of application z), the latter describes large categories of vulnerabilities (e.g., Access of Memory Location After End of Buffer).

2.1 Semantic Dimension

Semantic dimension refers to the natural meaning and interpretation of code tokens, treating them similarly to words in natural language. It aims to capture the meaning of code tokens.

For instance, consider two vulnerable methods in Listing 1 and 2 for the BigVul [26] dataset. The first method is responsible for freeing the memory associated with various components of the streamCG data structure, ensuring that any necessary cleanup or deallocation routines are called using the provided callback functions. The latter is designed to allocate memory for a TcpSession object.

While the primary objectives of these two methods differ significantly (one allocates memory, and the other deallocates it), they both include the keyword "stream" in their names. This repetition of "stream" in their names indicates their association with the manipulation of streaming data.

In the case of the streamFreeCG method, the word "stream" appears not only in the function's name but also in the parameter name. Furthermore, within the function's body, "stream" is used twice in the function calls, namely, "streamFreeNACK" and "streamFreeConsumer."

Likewise, in the function StreamTcpSessionPoolAlloc, the term "stream" is included once in its function name and twice in the function calls for "StreamTcpCheckMemcap" and "StreamTcpSessionClear." This shared usage of the term "stream" in their respective names and function calls signifies a certain semantic similarity between these two methods.

Note that this assessment of semantic similarity is based on the frequent usage of the word "stream" and the context in which these methods operate within the codebase.

Method "streamFreeCG" is vulnerable to "CVE-2018-12453", while method "StreamTcpSessionPoolAlloc" is vulnerable to "CVE-2018-14568". If we observe closely, we notice that although these two code samples share some semantic information, their context and syntax are different. This demonstrates the power of semantic dimension in this case while the two other dimensions are different.

Listing 1: streamFreeCG method to deallocate memory

```
1 void streamFreeCG(streamCG cg)
2 {
3     raxFreeWithCallback(cg->pel,(void() (void*))
4     streamFreeNACK);
5     raxFreeWithCallback(cg->consumers,(void() (void))
6     streamFreeConsumer);
7     zfree (cg);
8 }
```

Listing 2: A Semantically-similar method to streamFreeCG

```
1 static void *StreamTcpSessionPoolAlloc(void)
2 {
3     void *ptr = NULL;
4     if (StreamTcpCheckMemcap((uint32_t)sizeof
5     (TcpSession))== 0)
6         return NULL;
7     ptr = SCMalloc(sizeof(TcpSession));
8     if (unlikely (ptr == NULL))
9         return NULL;
10    StreamTcpSessionClear();
11    return ptr ;
12 }
```

2.2 Contextual Dimension

Contextual dimension considers the surrounding code, dependencies, and environment in which the code operates. It focuses on understanding the intended functionality and behavior of the code, going beyond its specific structure or grammar.

Consider the two examples from the CWE website in Listings 3 and 4 [63]. The first example above takes an IP address from a user, verifies that it is well-formed, and then looks up the hostname and finally copies it into a buffer.

The second one (Listing 4) applies an encoding procedure to an input string and stores it in a buffer. Both of them are writing data to buffer. We consider these two methods to be contextually similar. Both of them are vulnerable to CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer. We observe that although having the same objective, they vary semantically and syntactically. For

example, as opposed to the methods in the Listings 1 and 2, these two methods do not share words with a similar meaning and also their basic structures are syntactically different. This finding highlights the significance of the contextual similarity while the methods are not similar in other dimensions.

Listing 3: A vulnerable example from CWE website

```

1 void host_lookup(char *user_supplied_addr)
2 {
3     struct hostent *hp;
4     in_addr_t *addr;
5     char hostname[64];
6     in_addr_t inet_addr(const char *cp);

7     /* routine that ensures user_supplied_addr is in
8        the right format for conversion */

9     validate_addr_form(user_supplied_addr);
10    addr = 10     inet_addr(user_supplied_addr);
11    hp = gethostbyaddr(addr, 12 sizeof(struct
12    in_addr), AF_INET);
13    strcpy(hostname, hp->h_name);
14 }

```

Listing 4: Another vulnerable example from CWE that shares similar contextuality

```

1 char * copy_input(char * user_supplied_string )
2 {
3     int i, dst_index;
4     char *dst_buf = (char*)malloc(4* sizeof(char)*
5     MAX_SIZE);
6     if ( MAX_SIZE <= strlen(user_supplied_string)
7     {
8         die("user string too long, die evil hacker!");
9     }
10    dst_index = 0;
11    for ( i = 0; i < strlen ( user_supplied_string );
12    i++)
13    {
14        if( '&' == user_supplied_string [I] )
15        {
16            dst_buf[dst_index++] = '&';
17            dst_buf[dst_index++] = 'a';
18            dst_buf[dst_index++] = 'm';
19            dst_buf[dst_index++] = 'p';
20            dst_buf[dst_index++] = ';';
21        }
22        else if ('<' == user_supplied_string [I] )
23        {
24            /* encode to &lt; */
25        }
26        else
27            dst_buf[dst_index++] =
28            user_supplied_string [i];
29    }
30    return dst_buf;
31 }

```

2.3 Syntactic Dimension

Syntactic dimension refers to the structure, grammar, and arrangement of symbols in code. It emphasizes the correct formation and arrangement of tokens, keywords, operators, and other language constructs according to the defined rules and conventions of the language.

For example, consider the two methods from BigVul dataset in Listings 5 and 6 where the first one is vulnerable to "CWE-189:Numeric Errors" and the second one is vulnerable to "CWE-119:Improper Restriction of Operations within the Bounds of a Memory Buffer". The basic syntactical structure of the methods is the same.

Listing 5: A method from BigVul dataset

```

1 void PaymentRequest::NoUpdatedPaymentDetails()
2 {
3     spec_>RecomputeSpecForDetails();
4 }

```

Listing 6: A syntactically similar method from BigVul dataset

```

1 void InitPrefMembers()
2 {
3     settings_>InitPrefMembers()
4 }

```

Both of them share a similar signature, with each function having a void return type and not accepting any parameters. Both functions consist of only one line of code, invoking a method on a class member object. Although syntactically similar, they do not share any similar semantics or functionality. The first one, invokes the `RecomputeSpecForDetails()` method on the `spec_` object. On the other hand, the second function, `InitPrefMembers()`, calls the `InitPrefMembers()` method on the `settings_` object, suggesting the initialization of preference-related members.

The above-mentioned examples motivate us to incorporate these dimensions to facilitate vulnerability detection. We hypothesize (and later validate) that this multi-dimensional similarity analysis allows us to efficiently utilize the limited amount of past vulnerability data to more accurately reason about vulnerabilities in a piece of code.

Additionally, we conducted an initial experiment with the vulnerable records from the BigVul dataset to check how the neighboring information can also be utilized in order to detect if a piece of code is vulnerable or not. Neighboring information is fetched using the cosine similarity metric for embeddings in each dimension. More detail about neighbor calculation is presented in Section 3. We fetched the neighbor information for 8,740 vulnerable samples. We looked for the closest matching neighbor in each dimension. We experimented with both CWE and CVE. The first part of Figure 1 shows the Venn diagram for CVE. The blue, orange, and green refer to being neighbors in contextual, semantic, and

syntactic dimensions respectively. We stored the CVE and CWE information for the first closest neighbor in each dimension. We notice that the CVE for the top neighbor of 199 records is actually the same in all three dimensions. We further notice some records where the top neighbor shares the same CVE in only one dimension. However, in the majority of the cases, they share the same CVE for at least one dimension. Among 8,740 records, for 3650 records the top most similar record was not similar in any dimension. A similar trend is also observed when we consider CWE information in the second part of the diagram. Note that in this demonstration, we only considered the topmost neighbor. Our hypothesis was that considering the top n neighbors instead would improve the precision of our analysis in determining the vulnerability of a piece of code. Later in Section 4, we establish the validity of this hypothesis quantitatively and show that Vul-Sim outperforms the state-of-the-art vulnerability detection models.

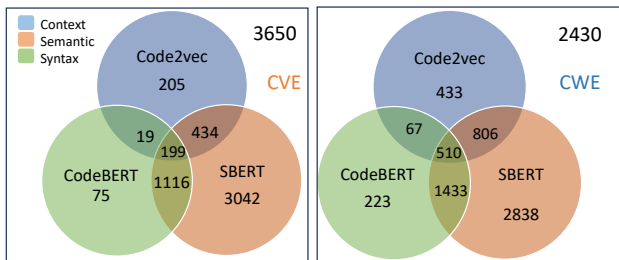


Figure 1: Venn Diagram of similarities of weaknesses and vulnerabilities in Context, Semantic and Syntax dimensions.

In the following subsections, we describe how we generated embeddings to gather information about these code dimensions. We explain how and why we selected some specific models for each dimension.

2.4 Semantic Embedding of Methods

Natural language processing (NLP) has made significant advancements with word embedding techniques like word2vec [58, 59] improving the ability to represent words in a continuous vector space. These embeddings capture the semantic meaning of words, allowing for similarity calculations between words based on their vector representations [68]. Previous efforts have explored applying word embedding techniques to source code, with promising results [17, 25, 42].

One pre-trained model based on transformers, called ‘‘Sentence-BERT’’ (SBERT), generates embeddings for sentences, including code snippets [77]. SBERT uses a shallow fully-connected neural network to model each sentence as a continuous vector, ensuring that sentences with similar contexts are represented closely in the vector space [6]. In the case of multiple-word contexts, SBERT takes the average of the word vectors and generates real-valued vectors. The

model learns distributed representations of training sentences and produces fixed-length, low-dimensional vector representations for any given sentence.

We use the SBERT model to transform the semantic meaning of terms within vulnerable and safe code fragments into embeddings. This transformation enables the measurement of semantic relevance among code snippet embeddings and the tagging of non-labeled instances. By treating code snippets as a natural language, the semantic model generates embeddings that capture the semantic properties of each term. The differences in semantics between vulnerable and non-vulnerable instances are then measured by computing the distance between their code embeddings.

Since we are interested in pure semantic information, and other dimensions are covered by different models in this work, we use the SBERT model. We opted not to use other complex alternatives such as GraphCodeBERT [38] since along with semantic information, they also well capture other information such as semantics as demonstrated by [57] that we intentionally do not want to utilize in this dimension. However, other semantic models such as word2vec [59] might also be utilized.

2.5 Contextual Embedding of Methods

Several code embedding methods have been developed to capture both semantic and structural information of code. Tree-based approaches, especially those based on abstract syntax trees (ASTs), have shown promise in improving vulnerability detection [5, 72].

To get the contextual properties, we use a recent approach called code2vec that leverages the AST structure of code to identify informative paths that capture the functionality of methods [6]. Code2vec uses an attention mechanism to compute a weighted average of the path vectors, allowing it to generate descriptive method names that represent the context of a method. The model is pre-trained on a large dataset of Java GitHub repositories, consisting of millions of samples, and has shown superior performance in various tasks, including vulnerability detection [21].

In their original work [6], the authors demonstrated the ability to detect method names for a given method based on functionality. Motivated by that work, we selected code2vec for our contextual space embeddings. As demonstrated by Alon et al. [6], vulnerable methods sharing the same functionality or context are therefore placed in closer space and we can leverage this information to detect if a piece of code is vulnerable or not.

2.6 Syntactic Embedding of Methods

Measuring syntactical properties of software artifacts as a similarity metric is common in the software engineering domain, including design and source code analysis ap-

plications [32, 34, 84, 85, 112]. This property is widely adopted in various applications, such as in code refactoring [64, 86, 99, 100], system remodularizations [29, 67, 97], mining features from object-oriented code [4], improving feature localization [73], and extracting code-relevant description sentences [12]. Structural measures are also utilized in software security to detect spam emails and ransomware applications [7, 35, 82].

We leveraged CodeBERT [28] model to capture the syntactic properties of source code. CodeBERT leverages the neural architecture of BERT, using stacked transformers in a bidirectional structure to capture long-range dependencies required for learning vulnerable code patterns. The model is pre-trained on both programming language (PL) and natural language (NL) data, learning distributed representations of both artifacts. It incorporates a hybrid objective function that includes a pre-training task of detecting replaced tokens. In CodeBERT architecture, the text-code encoder generates plausible tokens for masked positions, and the text-code decoder (discriminator) is trained to detect alternative token samples generated by the encoder. The application of CodeBERT for code analysis reveals latent patterns within software code and shows promise in facilitating various downstream tasks, including vulnerability detection.

CodeBERT captures syntactic information through its training process on a large dataset of code snippets. In their work, Karmakar et al. [43] demonstrated the syntactic property of CodeBERT by getting 89.45% accuracy for syntactic task - AST node tagging. Although basic BERT was also showing almost similar accuracy, we selected CodeBERT because it is specifically trained on source code. A similar observation was found by Wan et al. [104] where they observed that the syntax structure of code has been well preserved in different hidden layers of CodeBERT. A very recent study by Ma et al. [57] demonstrated several aspects of syntactic and semantic properties of several models such as CodeBERT, GraphCodeBERT, and several LLMs. Their study demonstrated the skill of CodeBERT in several syntactic tasks such as syntax pair node prediction, and token syntax tagging. It is worth mentioning that other models such as GraphCodeBERT, UnixCoder [37], and CodeT5 [108] were also having competitive performance. All these models including CodeBERT were performing their best in different layers and in general, all of them were performing well in syntactical tasks. However, models other than CodeBERT were also performing well in semantic tasks such as semantic relation prediction and semantic propagation. Since we wanted to emphasize syntactic properties, we utilized CodeBERT embeddings. Interestingly LLMs such as StarCoder [46], CodeLlama [79] and CodeT5+ [107] did not exhibit advantages over the pre-trained models in syntactic tasks.

3 VulSim’s Implementation

This section discusses the implementation of our multi-dimensional and neighbor-based classification approach for classifying vulnerable and safe methods, VulSim.

3.1 Dataset

We selected two commonly used C language datasets for evaluation. We focus on C because it is widely used in system building and its memory unsafety makes it more prone to large classes of vulnerabilities, notably memory corruption bugs that constitute around 70% of vulnerabilities [60, 101].

Table 1: Dataset statistics.

	Devign	BigVul
Number of Vul Methods	12,460	11,823
Number of Safe Methods	14,858	253,096
Number of Vul Ours	12,425	8,740
Number of Safe Ours	14,822	8,922

3.1.1 Devign

The Devign dataset [119] is commonly used in this domain [41, 74, 109]. The dataset comprises functions from the QEMU [2] and FFmpeg [1] open-source projects, labeled as vulnerable or non-vulnerable. The dataset is carefully constructed through manual classification of commits related to vulnerability fixes and extensive cross-validation, resulting in a balanced dataset of 12,460 vulnerable and 14,858 non-vulnerable methods. Devign is also utilized in the Microsoft CodexGLUE benchmark for evaluating vulnerability models [56]. Table 1 shows the total number of records in this dataset. In our work, we removed 71 records since Astminer [8, 44], the open-source tool we utilized was unable to generate AST representation for these records. All other records are analyzed without any modification.

3.1.2 BigVul

The BigVul dataset [26] offers a comprehensive collection of labeled vulnerability instances across various software systems, including vulnerabilities in different programming languages such as C. Its large-scale nature and diverse range of vulnerability types make it suitable for benchmarking and training vulnerability detection techniques. In our study, we utilized the BigVul dataset only to test the performance and generalizability of our proposed approach. Table 1 reports the total number of records in the BigVul dataset. Since it is not a balanced dataset, we took approximately an equal number of samples from the safe methods. Once again, we had to remove some of the samples because they had incomplete

information (lacked CVE and CWE information), which we leveraged in our analysis. Additionally, a small number of records are discarded because Astminer fails to run on them. In total, we used the remaining 8,740 vulnerable and 8,922 safe samples for our analysis.

3.2 Distributed Representation of Code

To build a consolidated space, we initially built three sets of embeddings, based on semantic, contextual, and syntactic properties of the C functions in the Devign dataset.

(i) Semantic Space: To leverage SBERT (Sentence-BERT) for generating embeddings to conduct semantic analysis on methods, we used the approach proposed by the authors [77]. In order to do the classification, we used SBERT as a classifier by utilizing the approach suggested in the related work [36]. Initially, we divided the dataset into a 90-10% ratio for training and testing purposes. Subsequently, we input the source code into a transformer-based binary classifier to classify it as either vulnerable or non-vulnerable.

To preserve the semantic space, we extracted the generated embeddings for the remaining part of our analysis.

To configure the model training process for classification, we used the default setup where the total number of epochs was set to 3, per device train batch size was 8, the batch size for evaluation was set to 20, the warmup steps for learning rate scheduler was 500, and weight decay was set to 0.01 since changing parameters like increasing warmup steps or epoch size did not improve the performance.

(ii) Contextual Space: The code2vec model [6] was originally trained in the Java programming language. Leveraging the open-source approach proposed by Coimbra et al. [21], we re-trained code2vec on code snippets, written in C, to account for the potential differences between the two languages.

For this, we initially generated the counterpart ASTs for the functions in our dataset, using Astminer. Among the 27,318 methods, Astminer was unable to generate the ASTs for 71 records. The remaining 27,247 AST representations were converted to the code2vec acceptable format to re-train the model for learning common patterns of C-related AST-based patterns.

As suggested in [21], we re-trained the network for 20 epochs, following the default hyper-parameters of the original code2vec with a batch size of 1,024, embedding size of 128, and the dropout rate of 0.25. Among the generated models we selected the model with the highest F1 score and used that model for validation purposes. Since the trained model [21] is already in the CodexGlue [56] leaderboard and shows good accuracy, we trusted their default parameters. Finally, we generated and stored the embeddings.

(iii) Syntactic Space: The original CodeBERT model [28] is initially trained in 6 programming languages, excluding C language. For this reason, we fine-tuned CodeBERT on code

snippets, written in C on the Devign dataset by following the instructions on CodexGLUE [55]. In this case, we retrained the model for 5 epochs with a block size of 400, training batch size of 32, and evaluation batch size of 64 with a learning rate of $2e^{-5}$ which was the default setup. Once again, we used the default parameter since the model is already at the top list of the CodexGlue benchmark.

For the construction of the syntactic space, we followed a similar approach by reading and storing the embeddings for our analysis.

The ultimate re-generated fine-tuned model generated an accuracy of 64.60% as shown in Figure 2 which is slightly higher than the original codeBERT model as the CodexGLUE leaderboard showed (62.08%). Given the three spaces we built, we then passed the embeddings to a feature generation component which calculates the relative closeness of the embeddings to training embeddings within each space individually.

3.3 Feature Generation

(i) Measuring Distance: In each space, we conducted individual pairwise similarity measurements between the embeddings of the functions. This similarity assessment was performed based on cosine similarity, a widely used approach utilized by several other efforts to calculate distance between vectors [6, 66, 110]. Cosine similarity can be determined using the following equation where A_i and B_i are the i -th component of vector A and B:

$$\text{Similarity}(A, B) = \frac{\sum_{i=1}^k A_i B_i}{\sqrt{\sum_{i=1}^k A_i^2} \sqrt{\sum_{i=1}^k B_i^2}}$$

The similarity value between two vectors A and B ranges from -1 to 1. When the angle between the vectors is smaller, the value of cosine similarity is larger which indicates that the values are closer to each other.

Given each set of the embeddings, we then calculated four $k \times k$ matrices of cosine similarity scores between all the embedding pairs, where k is the number of the embeddings present in the dataset. The matrices are symmetric as the top half of each matrix diagonally mirrors the bottom half.

(ii) Ranking and Scoring:

We developed a ranking mechanism by selecting the top n embeddings with the highest similarity score in each space, weights were calculated as:

$$\text{Score} = \sum_{i=0}^{n-1} (n-i)w_i$$

where n is the desired number of the closest neighbor embeddings to be considered for labeling the vectors. The i specifies the neighbors index, which are sorted in descending order according to their distance from a target vector. As such the

closest neighbor has a higher vote (larger contribution) in determining the label of the given vector. Here w_i denotes the cosine similarity value for the current record with i -th similar value. This is, in particular, important in scenarios, where the distance variance of n -top embeddings to the target embedding is large. This ranking mechanism was independently repeated for each individual space.

Given the ranking mechanism, we generated two scores for each test vector, one with selecting the n -top weak neighbors (*bad* score) and the other with safe neighbors (*good* score). To elaborate more, for a specific record, if 1st record ($i = 1$) is good and has a similarity value of w_i with the target, the score will be added to the good score. Similarly, for $i = 2$, if the record is bad, that score will be added to the bad score, and so on. As such, two scores were assigned to each vector, representing its similarity to vulnerable and safe code fragments. The scores were independently generated in each space, according to close neighbors of each particular space. These neighbor-based scores are finally leveraged to detect if a piece of code is vulnerable or not.

3.4 Classification

For classification, we adopted a decision tree-based classifier. This choice aimed to prevent the complexity of the classification algorithm from overshadowing the differences in classification capability among semantic, syntactic, and contextual properties. Additionally, we prioritized the simplicity and interpretability of the chosen model to gain clear insights into the prediction process. We configured the classifier with a Gini index, a maximum depth of 3, and a minimum of 5 samples per leaf.

For each item in the dataset, we fed the classifier with two sets of scores as we described in subsection 3.3. Based on these scores, the classifier classifies each sample as vulnerable or non-vulnerable. We trained the classifier with a ratio of 90%-10% for training and test sets.

For comparison purposes, we initially classified the embeddings based on individual vulnerable and safe scores in each individual space. Furthermore, we applied the same classifier to evaluate the combined power of all three property sets, creating a hybrid model, and considering all three properties simultaneously. In this case, we fed the classifier with 6 scores (2 from each space).

The classification assumption is that both, vulnerable and safe, code snippets will be mapped closer to the embeddings with similar features within at least one of the semantic, context, syntactic, or hybrid spaces.

4 Evaluation

To evaluate our approach, we performed experiments using different values of n , specifically 3 and 5, to explore the im-

part of different neighborhood sizes on the accuracy and performance of the technique.

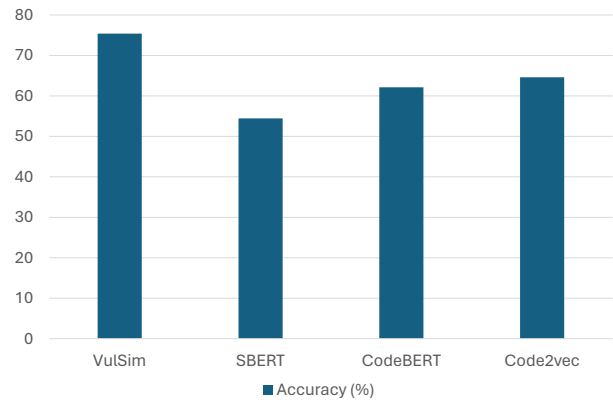


Figure 2: Accuracy of Original Models on Devign dataset.

4.1 RQ_1 : Evaluating the Impact of Utilizing Neighbors' Information from Multiple Dimensions

In this section, we assessed the ability of each space to detect vulnerabilities based on two key perspectives: the accuracy of classification and the uniqueness of identified vulnerability. This evaluation offers insights into the strengths and limitations of each space, providing an understanding of their individual contributions to vulnerability detection.

4.1.1 Detection Accuracy

Figure 2 represents the initial accuracy of the original models we leveraged for multiple dimensions. Table 3 illustrates the individual performance of each model (with neighboring information utilized) considered separately, their pair-wise combination, and the results of the hybrid model (VulSim) given the aggregated information.

By supplying the decision tree-based classifier with corresponding good and bad scores for each model, our context-based model demonstrated the highest average accuracy of 74% across various values of n . Notably, the semantic-only space also made a significant contribution, achieving accuracies of 61% and 65% for different n values. In contrast, the performance of the syntactic space was comparatively lower, yielding an accuracy of approximately 55%.

Moving on to the evaluation of double-dimensional spaces, the semantic-contextual model outperformed other models with an accuracy of 75%. Looking at VulSim results, it reveals that the overall performance of the multi-dimensional model, remained at 75% accuracy, showing that the inclusion of the syntactic property of the code did not result in a significant improvement in overall accuracy in this dataset, as presented in Table 3. It is worth mentioning that, although the

Table 2: Uniquely-detected instances in the Single-, Double-, and Hybrid-dimensional spaces are examined in our analysis.

		Within Group				Between Group				VulSim			
		n = 5		n = 3		n = 5		n = 3		n = 5		n = 3	
Group 1 (Single)	Semantic Syntactic contextual	Safe	Weak	Safe	Weak	Safe	Weak	Safe	Weak	Safe	Weak	Safe	Weak
		6	28	0	155	0	0	0	0	163	444	175	511
		37	28	59	1	19	19	34	1	6	765	5	820
		2	494	0	463	0	0	0	0	33	2	97	128
Group 2 (Double)	Semantic-contextual Semantic-Syntactic Syntactic-contextual	0	72	138	117	0	155	0	89	107	100	0	0
		18	164	25	89	0	0	0	0	33	2	97	128
		118	30	6	41	0	0	0	0	163	444	175	511
		Total		Safe:1,494				Vulnerable: 1,231					

Table 3: Accuracy of **Single**, **Double** and Multi spaces in classifying weak code for n = 5 and n = 3 of selected neighbors.

		Single Dimension					
		n = 5			n = 3		
		Semantic Space					
		Acc.	Prc.	Rec.	Acc.	Prc.	Rec.
Weak	Safe	60.92%	0.72	0.22	64.81%	0.69	0.40
			0.59	0.93		0.63	0.86
		Syntactic Space					
Weak	Safe	55.01%	0.52	0.06	54.75%	0.42	0.00
			0.55	0.96		0.55	1.00
		Contextual Space					
Weak	Safe	74.31%	0.76	0.62	74.09%	0.75	0.65
			0.73	0.84		0.74	0.82
		Double Dimensions					
		n = 5			n = 3		
		Semantic-Contextual Space					
		Acc.	Prc.	Rec.	Acc.	Prc.	Rec.
Weak	Safe	75.41%	0.77	0.65	75.34%	0.76	0.67
			0.75	0.84		0.75	0.82
		Semantic-Syntactic Space					
Weak	Safe	60.92%	0.72	0.22	74.09%	0.75	0.65
			0.59	0.93		0.74	0.82
		Syntactic-Contextual Space					
Weak	Safe	74.31%	0.76	0.62	64.81%	0.69	0.40
			0.73	0.84		0.63	0.86
		Multi-Dimensions (VulSim)					
		n = 5			n = 3		
		Semantic-Contextual-Syntactic Space					
		Acc.	Prc.	Rec.	Acc.	Prc.	Rec.
Weak	Safe	75.41%	0.77	0.65	75.34%	0.76	0.67
			0.75	0.84		0.75	0.82

accuracy, precision, and recall were comparatively lower, the following subsection will delve into the contributions of the syntactic space, demonstrating uniquely identified instances by each space and thereby underscoring the importance of incorporating multidimensional space. Moreover, since we incorporated multiple spaces in our final hybrid VulSim model, the lower precision and recall rate of syntactic dimension do

not have any negative impact since the hybrid model is exhibiting the best performance. Armed with this information, one may consider assigning higher weights to the more important dimensions to bias the model toward learning from those sets of attributes. Alternatively, one could opt to pass only the crucial dimensions to the final simple classifier. Additionally, it is important to note that this characteristic may not necessarily hold true in other datasets, as the nature of vulnerabilities and their relationships could vary among codebases.

In summary, if we look at Figure 2 once again, it provides a brief overview of how leveraging information from neighbors across different dimensions impacts accuracy. The accuracy of the original SBERT (54.46%), Code2vec (62.12%), and CodeBERT (64.6%) models we initially utilized to capture multiple dimensions is depicted along with VulSim (75.41%). From the diagram, it is evident that combining these three dimensions and incorporating neighbor information in VulSim leads to an increase in accuracy.

4.1.2 Uniqueness of Detected Instances

In addition to assessing the accuracy of the models, we conducted a thorough analysis of each space’s capability to detect unique instances that were misclassified by the other spaces. This investigation serves to support VulSim, which emphasizes the extraction of multiple sets of properties from a limited-size dataset to uncover additional rules.

Within Group: The analysis results are presented in Table 2. In the "Within Group" section, the number of correctly classified records that were uniquely identified by each single-dimension (Group 1) and double-dimension (Group 2) model is displayed out of a total of 1,231 weak entries in the test set. Among the single dimensions, the contextual space stands out by successfully identifying 494 and 463 unique instances (for five and three neighbors, respectively) compared to the other two single dimensions. This observation aligns with the highest accuracy of 74.31% and 74.09% achieved by this model, as shown in Table 3.

In the realm of double dimensions, the contextual-semantic space outperforms the other double-dimension spaces by iden-

tifying 117 unique instances that were missed by both the semantic-syntactic and syntactic-contextual spaces. Furthermore, when the number of neighbors is increased from three to five, the semantic-syntactic space emerges as the winner, identifying 164 unique vulnerabilities in the source code.

Between Group: The syntactic space identified 19 and 1 unique instances, respectively with 3 and 5 neighbors, in the between-group analysis. These instances were not only overlooked by the semantic and contextual single dimensions but also by the other three two-dimensional spaces that incorporated additional dimensional information. On the other hand, the semantic and contextual spaces, as individual entities, did not exhibit unique instances when compared to the more comprehensive double-dimensional spaces.

This observation is intriguing as it demonstrates that despite having the lowest accuracy among the single-space models, as shown in Table 3, the ability of the syntactic space to identify unique instances surpasses that of other models.

Hybrid: Significantly, the utilization of the hybrid model, which integrates similarity scores from all dimensions, resulted in the identification of 444, 765, and 2 records that were exclusively recognized when employing $n = 5$, a scenario in which the single-dimension models respectively semantic, syntactic, and contextual failed to recognize these instances despite possessing information about the same neighbors. This outcome not only underscores the effectiveness of leveraging neighbor information but also underscores the value of combining multiple attributes for precise classification of vulnerable and safe cases, as we proposed.

In the analysis of the semantic space, an interesting observation emerged: when considering a smaller number of closest neighbors ($n = 3$), a higher number of unique instances (155) were identified compared to expanding the neighbor information to $n = 5$, which only revealed 28 unique weak instances. This finding suggests that, in this specific dataset, vulnerabilities tend to have a less broad set of similar keywords within the code. In other words, vulnerabilities in the dataset seem to exhibit more distinct semantic patterns when a smaller neighborhood is considered, indicating that their keyword associations are not as widely shared as when a larger neighborhood is taken into account.

To answer RQ_1 , we conclude that consolidating information from distinct spaces achieved the highest accuracy in detecting weak and safe methods in C code. While certain dimensions exhibit lower accuracy, they still enable the model to identify unique examples that were missed by others. This underscores the importance of considering not only accuracy but also the ability of each space to detect instances that may be overlooked by other spaces. Thus, our results emphasize the value of incorporating multiple spaces to achieve a more comprehensive and effective detection approach as implemented in VulSim.

4.2 RQ_2 : Evaluating the Effectiveness of Approach vs. State-of-the-art (SOTA)

In this research question, we aim to evaluate VulSim in terms of two key aspects: the accuracy of detection and, also generalizability of the model, by comparing the hybrid model to the current state of the art. As such, we can assess the performance of our approach and understand its effectiveness in accurately detecting vulnerabilities as well as its ability to generalize to unseen codebases different from that seen during training.

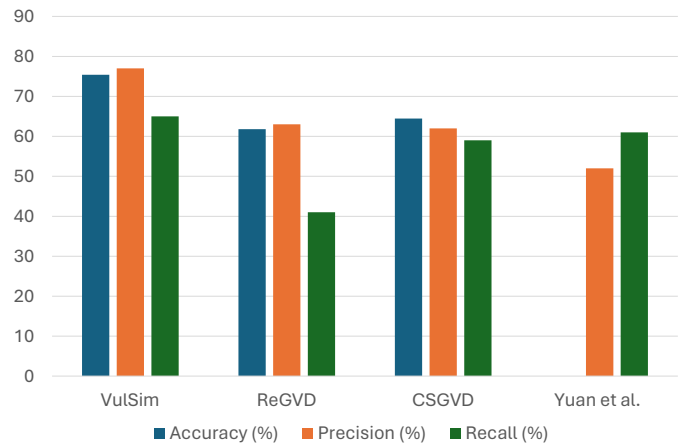


Figure 3: Accuracy of state-of-the-art vulnerability detection models on **Devign** dataset.

4.2.1 Detection Accuracy

Table 4 presents the leaderboard of the Microsoft CodexGLUE benchmark [56] on the Devign dataset for defect detection. Each model in the table is accompanied by a brief description and its corresponding accuracy. The leaderboard includes a total of 15 models, with the lowest accuracy recorded at 59.37%. The highest-performing model, UniXcoder-nine-MLP, achieves an accuracy of 69.29%. Interestingly, the amalgamated space exhibited a superior accuracy of 75.41% when incorporating information from the five nearest neighbors in the hybrid space.

In addition to the models listed on the leaderboard, we conducted experiments with the original transformer-based classifiers, and the accuracy of these models are reported in Figure 2. Notably, these values are slightly different than the accuracy recorded on the leaderboard for the same dataset.

We additionally conducted a comparison with SOTA vulnerability detection methods (Figure 3). We compared our approach with ReGVD [69], CSGVD [94] and the approach proposed by Yuan et al. [113]. ReGVD is a programming-language-independent technique that utilizes graph neural networks to detect vulnerability. ReGVD considers a mixture

between the sum and max poolings to produce a graph embedding for the source code. This graph embedding is then fed into a single fully connected layer followed by a softmax layer and predicts the vulnerabilities in the source code. Another model, CSGVD accepts the control flow graph of the source code as input and deals the vulnerability detection as a graph classification task. Yuan et al. [113] developed a Behavior Graph Model to connect the behaviors of different functions and enhance the detection ability of existing DL-based methods using this information. We additionally attempted to implement VulChecker [61], a DL framework that detectors instruction- and line-level vulnerability and Devign, that used a gated graph neural network model with the Conv module for graph-level classification. VulChecker is trained on a different dataset and we were unable to implement their model due to some reproducibility errors. We also faced reproducibility issues with DeVign [119], and also confirmed from the raised issue on their GitHub repository [22]. Similarly, another SOTA tool ReVeal was not reproducible, as their data was no longer available, which we also verified from the GitHub issues [78]. Attempt to re-implement SOTA fine-tuning LLM based model [83] also failed due to a replication error ("huggingface_hub.errors.HFValidationError: Repo id must be in the form 'repo_name' or 'namespace/repo_name': '/home/ma-user/modelarts/inputs/model_2/'. Use repo_type argument if needed."). We did not find the model in the GitHub repository that we had to provide in order to successfully run their code.

As can be observed in Figure 3, VulSim outperformed all other baseline models in terms of accuracy, precision, and recall where the closest accuracy was detected by CSGVD (64.46%). The accuracy of ReGVD, mentioned by the authors was 63.69%, and the recreated accuracy for ReGVD in our experiment was 61.79%. Yuan et al. [113] did not provide any accuracy measurement in the paper. However, VulSim performed better than their approach in terms of both precision (77% compared to 52%) and recall (65% compared to 62%).

4.2.2 Detection Generalizability

In order to evaluate the applicability and versatility of our approach, we subjected VulSim to a completely new and previously unseen dataset, BigVul, which was not utilized during the training process. The objective was to evaluate how well our model could perform on this novel dataset, which differed significantly from the Devign dataset used for training. This experiment allowed us to gain insights into the model's capacity to generalize its learning and effectively handle real-world scenarios beyond the data with which it was familiar.

The results of the experiment are presented in Table 5. Despite the substantial change in the dataset, our model displayed a high performance by accurately identifying 85% of the weak methods within the different-style code, which serves a completely different application with a precision of 53%. The recall increased significantly from 65% to 85% (for

Table 4: Accuracy of state-of-the-art learning-based models according to leader board from the CodeXGLUE Benchmark on Devign dataset.

Model	Architecture	Acc%
VulSim	Multi-Dimensional neighbor-based	75.41
UniXcoder-nine-MLP	Not accessible	69.29
CoText [75]	T*-based encoder-decoder	66.60
C-BERT [10]	AST-based bidirect T*-based	65.45
A-BERT	Adversary-specific T*-based	65.37
RefactorBERT	Refactoring-specific bidirect T*-based	65.08
VulBERTa-MLP	Vul-specific BERT&MLP	64.75
VulBERTa-CNN	Vulnerability-specific BERT&CNN	64.42
ContraBERT_C	Not accessible	64.17
ContraBERT_G	Not accessible	63.32
PLBART [3]	Bidirectional T*-based	63.18
RoBERTa [55]	Bidirectional T*-based	61.05
TextCNN [55]	CNN-based NL pre-trained	60.69
BiLSTM [55]	Bidirectional LSTM-based	59.37
T*: Transformer-based		

Table 5: Generalizability of approach in classifying weak code for $n = 5$ and $n = 3$ of selected neighbors on unseen data.

	VulSim					
	$n = 5$			$n = 3$		
	Acc.	Prc.	Rec.	Acc.	Prc.	Rec.
Weak		0.53	0.85		0.52	0.80
Safe	55.86%	0.66	0.28	54.33%	0.61	0.30

5 neighbors) and similarly from 67% to 80% (for 3 neighbors). This change in the testing dataset led to a notable enhancement in the retrieval of vulnerable code detection rather than a deterioration. However, it is important to note that more number of false positives (lower precision) was the trade-off for the improved recall.

The performance evaluation of the models in the CodexGLUE benchmark was conducted based on training and testing on the Devign dataset. Initially, our intention was to test the top models on the benchmark on the same BigVul dataset to facilitate comparison. However, we encountered limitations in accessing the source code and data for some of the top-performing models, making their implementation infeasible. Specifically, CoText could not be re-implemented due to library issues.

Among the models, we were able to evaluate, as can be observed from Table 6, VulBERTa-MLP reported an accuracy of 64.75%. Upon regeneration, our results showed a comparable accuracy of 64.71%. However, VulSim outperformed VulBERTa-MLP in terms of recall rate to detect vulnerable code (85% compared to 37%) and overall accuracy (55% compared to 49.72%). Vulberta-CNN reported an accuracy of 64.42. Upon re-generation using their provided code, we achieved 53.60% accuracy which is around 11% lower than the reported value. While testing on Big-Vul dataset, it gave an accuracy of around 54.6% which is similar to

Table 6: Generalizability of approach in classifying weak code for other models on BigVul dataset

	Reported	Re-generated		Tested on BigVul			
	Acc. (%)	Acc. (%)	Prc.	Rec.	Acc. (%)	Prc.	Rec.
VulBERTa-MLP	64.75	64.71	0.65	0.51	49.72	0.49	0.37
VulBERTa-CNN	64.42	53.60	0.48	0.12	54.60	0.50	0.13
ReGVD	63.69	61.79	0.63	0.41	48.91	0.48	0.34

VulSim. However, in terms of recall, it performed poorly compared to VulSim (13% compared to 85% of Vulsim), which means it did not do a good job of recognizing vulnerable records successfully. Additionally, we compared our results with ReGVD [69]. ReGVD achieved a lower recall rate (33.90%) and overall accuracy (48.91%) compared to VulSim and VulBERTa.

This outcome signifies the robustness and adaptability of VulSim, as it successfully recognized a significant portion of vulnerable methods even when faced with previously unseen and diverse code structures. These findings underscore the potential of VulSim in effectively detecting weak methods in various code contexts and highlight the importance of further optimizing the precision to enhance the overall performance and reliability of the model.

4.3 Report

In addition to classifying methods as safe or vulnerable, our framework generates a comprehensive report containing information about the two closest neighbors of a given method in each space to provide insightful context for human-in-the-loop evaluation. An example report generated by VulSim is illustrated in Figure 4 in Appendix A. This report assists in manually checking a function by providing a brief description and the top two neighbors in each dimension, along with descriptions to aid vulnerability detection.

5 Threats to Validity

Potential biases can arise from the usage of a C-based dataset. While the proposed approach has demonstrated promising results, establishing its generalizability to other programming languages requires further experimentation. Nevertheless, a noteworthy advantage of our approach is its capability to account for semantic properties even in scenarios where the code's structure and syntax significantly differ. This characteristic enables the model to make reasonable decisions even in the presence of substantial code variations. To address this threat, we can minimize its impact by assigning higher weights to the semantic properties when transferring models trained in one language to be applied in an environment using a different programming language.

Furthermore, achieving consistent functionality sharing among various vulnerabilities is challenging because of gran-

ularity concerns. In some cases, a single method may not fully encapsulate the intended functionality, leading to higher-level functionality that spans across multiple methods. This situation necessitates the identification of all relevant scattered methods to accurately determine functional similarity among vulnerabilities.

Conversely, opting for a file-level comparison may result in the selection of files that contain small and irrelevant components unrelated to the actual functionality of the file, such as a method for serializing the output [27]. This issue highlights the importance of striking a balance in selecting the level of granularity for functionality representation to ensure that relevant and meaningful similarities are captured while avoiding the inclusion of extraneous and unrelated elements.

One limitation arose from the token size constraints of transformer-based models, which affected the generation of CodeBERT embeddings. To accommodate the token size limit of 512, we had to truncate the tokens, which may have resulted in the loss of some syntactic information from the code snippets. Furthermore, during the fine-tuning process of the code2vec model for the DeVign dataset, 71 records were discarded as they could not be represented in AST format. Although these discarded records only account for 0.002% of the entire dataset, we recognize the potential impact on the representativeness of the model.

To mitigate the influence of these limitations, we carefully considered the impact on the overall system's performance. While the token truncation in CodeBERT embeddings might have affected the full context representation, we selected an appropriate token size to balance model performance and computational efficiency. Additionally, the discarded records were a very small fraction of the samples in the dataset.

6 Related Works

In the field of vulnerability detection, two primary approaches are widely used: static and dynamic analysis of the source code. While static analysis examines the source code without executing it, dynamic analysis involves executing the program with specific input data. Hybrid analysis, which combines both static and dynamic approaches, is also commonly practiced.

Machine learning (ML) techniques have been leveraged for vulnerability detection, categorized into four main types as proposed by Ghaffarian and Shahriari [33]: prediction models based on software metrics, anomaly detection ap-

proaches, vulnerable code pattern recognition, and miscellaneous approaches. The first category utilizes source code files, object-oriented classes, and binary components to train ML models for vulnerability detection. For instance, Moshtari et al. [65] proposed a semi-analysis framework for within-project and cross-project vulnerability prediction. Anomaly detection methods, such as Chucky proposed by Yamaguchi et al. [111], identify software defects by detecting unusual patterns in the source code. Pattern recognition techniques, like the N-gram text-mining approach used by Pang et al. [71], analyze large data sets to detect source code vulnerabilities. Transformer-based deep learning approaches have also been utilized for vulnerability detection, with models like Vulberta proposed by Hanif and Maffei [39] that pre-trains RoBERTa with a custom tokenization pipeline.

Several efforts contributed to vulnerability detection. For instance, Du et al. [24] proposed a lightweight framework called Leopard, which assesses vulnerabilities through program metrics and ranking mechanisms. Li et al. [47] introduced IVDetect, an interpretable vulnerability detector that provides vulnerability interpretations based on vulnerable statements and their surroundings.

Other efforts have explored incorporating semantic information in vulnerability detection, as seen in efforts by Wang et al. [106], Choi et al. [20], Li et al. [50], Liu et al. [53], Zhou et al. [119], Li et al. [49], and Sun et al. [93]. Chan et al. [14] and Zhao et al. [117] explored vulnerability detection during code editing and utilized function fingerprints and code differences, respectively. Le and Babar [45] used real-world data to investigate ML models for automating function-level vulnerability assessment tasks such as predicting Common Vulnerability Scoring System. Benjamin et al. [91] reproduced 9 State-of-the-Art vulnerability detection models and answered research questions in three areas, model capabilities, training data, and model interpretation.

Cai et al. [11] proposed a vulnerability detection method based on deep learning with complex network analysis and subgraph partition. Another recent work is VDDA [15] where the authors utilized deep learning and attention mechanism-based combined architecture for vulnerability detection. Mirsky et al. [61] proposed VulChecker, DL framework that detectors instruction and line level vulnerability. Their methodology lacks consideration for all vulnerability types in general. Instead, it focuses on 5 Common Weakness Enumeration (CWE) categories, employing separate models to detect vulnerabilities within each of these specific categories. Cheng et al. [18] proposed Path-Sensitive Code Embedding utilizing a pre-trained value-flow path encoder via self-supervised contrastive learning. Wang et al. [105] developed a multi-relational, gated graph neural network vulnerability detection by combining probabilistic learning and statistical assessment to develop a “mixture-of-experts” approach to address the shortage of vulnerable training code samples. They further exploited transfer learning to port vul-

nerability detection models across programming languages.

While our approach focused on detecting vulnerabilities at the function-level granularity, other researchers like Hin et al. [33] and Li et al. [39], Fu and fu2022linevul [30], Dong et al. [23] explored the detection of vulnerabilities at the statement level. Zhang et al. [115] also proposed statement-level vulnerability detection approach CPVD that combines Graph Attention Network and Domain Adaptation Representation Learning to detect vulnerability in source code. Additionally, [48, 51, 87, 96] worked on code gadget/slice-level vulnerability detection.

Recent work focuses on vulnerability detection based on Large Language Models (LLM)-based approaches [19, 31, 54, 76, 83]. As demonstrated by Steenhoek et al. [92] LLMs generally struggled with vulnerability detection tasks. A recent work by Shestov et al. [83] demonstrated better performance of fine-tuned LLM compared to CodeBERT-like models. However, more research needs to be done to compare their work with SOTA vulnerability detection tools [9, 88–90, 103] to better validate such approaches.

In contrast to the aforementioned efforts, our approach aims to utilize multiple properties of source code to maximize the potential of detecting vulnerabilities. By considering various aspects of code representation and neighbor information, we seek to enhance the effectiveness of vulnerability detection in VulSim.

7 Conclusion and Future Works

In this paper, we introduce VulSim, a vulnerability detection tool. Through the analysis of diverse code embeddings, we initially hypothesize and subsequently validate that incorporating multiple dimensions of code properties (i.e., syntactic, semantic, and contextual) alongside neighboring information enhances vulnerability detection. Experimental results indicate, that VulSim outperforms other SOTA vulnerability detection models and exhibits good results when tested on unseen data. Future work includes applying our approach to identify vulnerabilities in other languages as well as studying the impact of applying it in a domain-specific fashion (e.g., to certain types of applications or certain types of vulnerabilities).

8 Acknowledgement

The work in this paper was partially funded by the Office of Naval Research (ONR) (Grant#: G2A62826).

References

- [1] Ffmpeg. <https://github.com/FFmpeg/FFmpeg>, note = Accessed: 2023-04-04.
- [2] Qemu. <https://github.com/qemu/qemu>, note = Accessed: 2023-04-04.
- [3] AHMAD, W. U., CHAKRABORTY, S., RAY, B., AND CHANG, K.-W. Unified pre-training for program understanding and generation, 2021.
- [4] AL-MSIE' DEEN, R., SERIAI, A.-D., HUCHARD, M., URTADO, C., AND VAUTIER, S. Mining features from the object-oriented source code of software variants by combining lexical and structural similarity. In *2013 IEEE 14th International Conference on Information Reuse Integration* (2013), pp. 586–593.
- [5] ALADICS, T., HEGEDŰS, P., AND FERENC, R. An ast-based code change representation and its performance in just-in-time vulnerability prediction. In *International Conference on Software Technologies* (2022), Springer, pp. 169–186.
- [6] ALON, U., ZILBERSTEIN, M., LEVY, O., AND YAHAV, E. code2vec: Learning distributed representations of code. *ACM on Programming Languages* 3, POPL (2019), 1–29.
- [7] ALZHRANI, A., ALSHEHRI, A., ALSHAHRANI, H., ALHARTHI, R., FU, H., LIU, A., AND ZHU, Y. Randroid: structural similarity approach for detecting ransomware applications in android platform. In *2018 IEEE International Conference on Electro/Information Technology (EIT)* (2018), IEEE, pp. 0892–0897.
- [8] ASTMINER. `astminer`. <https://github.com/JetBrains-Research/astminer>, 2007. Accessed: 2023-28-07.
- [9] BIGELOW, D., HOBSON, T., RUDD, R., STREILEIN, W., AND OKHRAVI, H. There's a hole in the bottom of the c: On the effectiveness of allocation protection. In *Proceedings of the IEEE Secure Development Conference (SecDev18)* (Oct 2018).
- [10] BURATTI, L., PUJAR, S., BORNEA, M., MCCARLEY, S., ZHENG, Y., ROSSIELLO, G., MORARI, A., LAREDO, J., THOST, V., ZHUANG, Y., AND DOMENICONI, G. Exploring software naturalness through neural language models, 2020.
- [11] CAI, W., CHEN, J., YU, J., AND GAO, L. A software vulnerability detection method based on deep learning with complex network analysis and subgraph partition. *Information and Software Technology* 164 (2023), 107328.
- [12] CAO, Y., ZOU, Y., AND XIE, B. Extracting code-relevant description sentences based on structural similarity. In *Proceedings of the 11th Asia-Pacific Symposium on Internetware* (New York, NY, USA, 2019), Internetware '19, Association for Computing Machinery.
- [13] CHAKRABORTY, S., KRISHNA, R., DING, Y., AND RAY, B. Deep learning based vulnerability detection: Are we there yet? *IEEE Transactions on Software Engineering* 48, 9 (2022), 3280–3296.
- [14] CHAN, A., KHARKAR, A., MOGHADDAM, R. Z., MOHYLEVSKYY, Y., HELYAR, A., KAMAL, E., ELKAMHAWY, M., AND SUNDARESAN, N. Transformer-based vulnerability detection in code at edittime: Zero-shot, few-shot, or fine-tuning? *arXiv preprint arXiv:2306.01754* (2023).
- [15] CHANG, J., MA, Z., CAO, B., AND ZHU, E. Vdda: An effective software vulnerability detection model based on deep learning and attention mechanism. In *2023 26th International Conference on Computer Supported Cooperative Work in Design (CSCWD)* (2023), IEEE, pp. 474–479.
- [16] CHEN, Y., DING, Z., ALOWAIN, L., CHEN, X., AND WAGNER, D. Diversevul: A new vulnerable source code dataset for deep learning based vulnerability detection. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses* (2023), pp. 654–668.
- [17] CHEN, Z., AND MONPERRUS, M. A literature study of embeddings on source code. *arXiv preprint arXiv:1904.03061* (2019).
- [18] CHENG, X., ZHANG, G., WANG, H., AND SUI, Y. Path-sensitive code embedding via contrastive learning for software vulnerability detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (2022), pp. 519–531.
- [19] CHESHKOV, A., ZADOROZHNY, P., AND LEVICHEV, R. Evaluation of chatgpt model for vulnerability detection. *arXiv preprint arXiv:2304.07232* (2023).
- [20] CHOI, M.-J., JEONG, S., OH, H., AND CHOO, J. End-to-end prediction of buffer overruns from raw source code via neural memory networks. *arXiv preprint arXiv:1703.02458* (2017).
- [21] COIMBRA, D., REIS, S., ABREU, R., PĂȘĂREANU, C., AND ERDOGMUS, H. On using distributed representations of source code for the detection of c security vulnerabilities. *arXiv preprint arXiv:2106.01367* (2021).
- [22] DEVIGNISSUE. `Devignissue`. <https://github.com/epicosy/devign/issues>, 2007. Accessed: 2023-07-29.
- [23] DONG, Y., TANG, Y., CHENG, X., YANG, Y., AND WANG, S. Sedsvd: Statement-level software vulnerability detection based on relational graph convolutional network with subgraph embedding. *Information and Software Technology* 158 (2023), 107168.
- [24] DU, X., CHEN, B., LI, Y., GUO, J., ZHOU, Y., LIU, Y., AND JIANG, Y. Leopard: Identifying vulnerable code for vulnerability assessment through program metrics. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)* (2019), IEEE, pp. 60–71.
- [25] EFSTATHIOU, V., AND SPINELLIS, D. Semantic source code models using identifier embeddings. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)* (2019), IEEE, pp. 29–33.
- [26] FAN, J., LI, Y., WANG, S., AND NGUYEN, T. N. Ac/c++ code vulnerability dataset with code changes and cve summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories* (2020), pp. 508–512.
- [27] FANG, C., LIU, Z., SHI, Y., HUANG, J., AND SHI, Q. Functional code clone detection with syntax and semantics fusion learning. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (2020), pp. 516–527.
- [28] FENG, Z., GUO, D., TANG, D., DUAN, N., FENG, X., GONG, M., SHOU, L., QIN, B., LIU, T., JIANG, D., ET AL. Codebert: A pre-trained model for programming and natural languages.
- [29] FOKAEFS, M., TSANTALIS, N., STROULIA, E., AND CHATZIGEORGIOU, A. Jdeodorant: identification and application of extract class refactorings. In *2011 33rd International Conference on Software Engineering* (2011), IEEE, pp. 1037–1039.
- [30] FU, M., AND TANTITHAMTHAVORN, C. Linevul: A transformer-based line-level vulnerability prediction. In *Proceedings of the 19th International Conference on Mining Software Repositories* (2022), pp. 608–620.
- [31] FU, M., TANTITHAMTHAVORN, C. K., NGUYEN, V., AND LE, T. Chatgpt for vulnerability detection, classification, and repair: How far are we? In *2023 30th Asia-Pacific Software Engineering Conference (APSEC)* (2023), IEEE, pp. 632–636.
- [32] GARLAN, D., SCHMERL, B., AND CHENG, S.-W. Software architecture-based self-adaptation. In *Autonomic computing and networking*. Springer, 2009, pp. 31–55.
- [33] GHAFARIAN, S. M., AND SHAHRIARI, H. R. Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. *ACM Computing Surveys (CSUR)* 50, 4 (2017), 1–36.
- [34] GODFREY, M. W., AND ZOU, L. Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering* 31, 2 (2005), 166–181.

- [35] GOMES, L. H., CASTRO, F. D., ALMEIDA, V. A., ALMEIDA, J. M., ALMEIDA, R. B., AND BETTENCOURT, L. M. Improving spam detection based on structural similarity. *SRUTI 5* (2005), 12–12.
- [36] GOOGLE. Sbert vs. data2vec on text classification.
- [37] GUO, D., LU, S., DUAN, N., WANG, Y., ZHOU, M., AND YIN, J. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850* (2022).
- [38] GUO, D., REN, S., LU, S., FENG, Z., TANG, D., LIU, S., ZHOU, L., DUAN, N., SVYATKOVSKIY, A., FU, S., ET AL. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366* (2020).
- [39] HANIF, H., AND MAFFEIS, S. Vulberta: Simplified source code pre-training for vulnerability detection. *arXiv preprint arXiv:2205.12424* (2022).
- [40] HANIF, H., NASIR, M. H. N. M., AB RAZAK, M. F., FIRDAUS, A., AND ANUAR, N. B. The rise of software vulnerability: Taxonomy of software vulnerabilities detection and machine learning approaches. *Journal of Network and Computer Applications 179* (2021), 103009.
- [41] JAIN, R., GERVASONI, N., NDHLOVU, M., AND RAWAT, S. A code centric evaluation of c/c++ vulnerability datasets for deep learning based vulnerability detection techniques. In *Proceedings of the 16th Innovations in Software Engineering Conference* (2023), pp. 1–10.
- [42] KANADE, A., MANIATIS, P., BALAKRISHNAN, G., AND SHI, K. Learning and evaluating contextual embedding of source code. In *International Conference on Machine Learning* (2020), PMLR, pp. 5110–5121.
- [43] KARMAKAR, A., AND ROBBES, R. What do pre-trained code models know about code? In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2021), IEEE, pp. 1332–1336.
- [44] KOVALENKO, V., BOGOMOLOV, E., BRYKSIN, T., AND BACCHELLI, A. Pathminer: a library for mining of path-based representations of code. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)* (2019), IEEE, pp. 13–17.
- [45] LE, T. H. M., AND BABAR, M. A. On the use of fine-grained vulnerable code statements for software vulnerability assessment models. In *Proceedings of the 19th International Conference on Mining Software Repositories* (2022), pp. 621–633.
- [46] LI, R., ALLAL, L. B., ZI, Y., MUENNIGHOFF, N., KOCETKOV, D., MOU, C., MARONE, M., AKIKI, C., LI, J., CHIM, J., ET AL. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023).
- [47] LI, Y., WANG, S., AND NGUYEN, T. N. Vulnerability detection with fine-grained interpretations. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2021), pp. 292–303.
- [48] LI, Z., ZOU, D., XU, S., JIN, H., ZHU, Y., AND CHEN, Z. Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing 19*, 4 (2021), 2244–2258.
- [49] LI, Z., ZOU, D., XU, S., JIN, H., ZHU, Y., ZHANG, Y., CHEN, Z., AND LI, D. Vuldeelocator: A deep learning-based system for detecting and locating software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing* (2021).
- [50] LI, Z., ZOU, D., XU, S., OU, X., JIN, H., WANG, S., DENG, Z., AND ZHONG, Y. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681* (2018).
- [51] LI, Z., ZOU, D., XU, S., OU, X., JIN, H., WANG, S., DENG, Z., AND ZHONG, Y. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681* (2018).
- [52] LIN, G., WEN, S., HAN, Q.-L., ZHANG, J., AND XIANG, Y. Software vulnerability detection using deep neural networks: a survey. *Proceedings of the IEEE 108*, 10 (2020), 1825–1848.
- [53] LIU, S., DIBAEI, M., TAI, Y., CHEN, C., ZHANG, J., AND XIANG, Y. Cyber vulnerability intelligence for internet of things binary. *IEEE Transactions on Industrial Informatics 16*, 3 (2019), 2154–2163.
- [54] LU, G., JU, X., CHEN, X., PEI, W., AND CAI, Z. Grace: Empowering llm-based software vulnerability detection with graph structure and in-context learning. *Journal of Systems and Software 212* (2024), 112031.
- [55] LU, S., GUO, D., REN, S., HUANG, J., SVYATKOVSKIY, A., BLANCO, A., CLEMENT, C., DRAIN, D., JIANG, D., TANG, D., LI, G., ZHOU, L., SHOU, L., ZHOU, L., TUFANO, M., GONG, M., ZHOU, M., DUAN, N., SUNDARESAN, N., DENG, S. K., FU, S., AND LIU, S. Codexglue: A machine learning benchmark dataset for code understanding and generation, 2021.
- [56] LU, S., GUO, D., REN, S., HUANG, J., SVYATKOVSKIY, A., BLANCO, A., CLEMENT, C. B., DRAIN, D., JIANG, D., TANG, D., LI, G., ZHOU, L., SHOU, L., ZHOU, L., TUFANO, M., GONG, M., ZHOU, M., DUAN, N., SUNDARESAN, N., DENG, S. K., FU, S., AND LIU, S. Codexglue: A machine learning benchmark dataset for code understanding and generation. *CoRR abs/2102.04664* (2021).
- [57] MA, W., LIU, S., ZHAO, M., XIE, X., WANG, W., HU, Q., ZHANG, J., AND LIU, Y. Unveiling code pre-trained models: Investigating syntax and semantics capacities. *ACM Transactions on Software Engineering and Methodology*.
- [58] MIKOLOV, T., CHEN, K., CORRADO, G., AND DEAN, J. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).
- [59] MIKOLOV, T., SUTSKEVER, I., CHEN, K., CORRADO, G. S., AND DEAN, J. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems* (2013), pp. 3111–3119.
- [60] MILLER, M. Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape (feb 2019). URL <https://github.com/microsoft/MSRC-Security-Research>.
- [61] MIRSKY, Y., MACON, G., BROWN, M., YAGEMANN, C., PRUETT, M., DOWNING, E., MERTOGUNO, S., AND LEE, W. Vulchecker: Graph-based vulnerability localization in source code. In *31st USENIX Security Symposium, Security 2022* (2023).
- [62] MITRE. Common vulnerabilities and exposures. <https://cve.mitre.org/cve/>, 2005. Accessed: 2023-10-20.
- [63] MITRE. Common weakness enumeration. <https://cwe.mitre.org/>, 2005. Accessed: 2020-10-20.
- [64] MOGHADAM, I. H., AND CINNÉIDE, M. Ó. Automated refactoring using design differencing. In *2012 16th European Conference on Software Maintenance and Reengineering* (2012), IEEE, pp. 43–52.
- [65] MOSHTARI, S., SAMI, A., AND AZIMI, M. Using complexity metrics to improve software security. *Computer Fraud & Security 2013*, 5 (2013), 8–17.
- [66] MOSOLYÓ, B., VÁNDOR, N., HEGEDŰS, P., AND FERENC, R. A line-level explainable vulnerability detection approach for java. In *Computational Science and Its Applications – ICCSA 2022 Workshops* (Cham, 2022), O. Gervasi, B. Murgante, S. Misra, A. M. A. C. Rocha, and C. Garau, Eds., Springer International Publishing, pp. 106–122.
- [67] NASEEM, R., MAQBOOL, O., AND MUHAMMAD, S. Improved similarity measures for software clustering. In *2011 15th European Conference on Software Maintenance and Reengineering* (2011), IEEE, pp. 45–54.
- [68] NASEEM, U., RAZZAK, I., KHAN, S. K., AND PRASAD, M. A comprehensive survey on word representation models: From classical to state-of-the-art word representation language models. *Transactions on Asian and Low-Resource Language Information Processing 20*, 5 (2021), 1–35.

- [69] NGUYEN, V.-A., NGUYEN, D. Q., NGUYEN, V., LE, T., TRAN, Q. H., AND PHUNG, D. Regvd: Revisiting graph neural networks for vulnerability detection. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings* (2022), pp. 178–182.
- [70] OKHRAVI, H. A cybersecurity moonshot. *IEEE Security & Privacy* 19, 3 (2021), 8–16.
- [71] PANG, Y., XUE, X., AND NAMIN, A. S. Predicting vulnerable software components through n-gram analysis and statistical feature selection. In *14th International Conference on Machine Learning and Applications* (2015), IEEE, pp. 543–548.
- [72] PARTENZA, G., AMBURGEY, T., DENG, L., DEHLINGER, J., AND CHAKRABORTY, S. Automatic identification of vulnerable code: Investigations with an ast-based neural network. In *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)* (2021), IEEE, pp. 1475–1482.
- [73] PENG, X., XING, Z., TAN, X., YU, Y., AND ZHAO, W. Improving feature location using structural similarity and iterative graph mapping. *Journal of Systems and Software* 86, 3 (2013), 664–676.
- [74] PHAN, L., TRAN, H., LE, D., NGUYEN, H., ANIBAL, J., PELTEKIAN, A., AND YE, Y. Cotext: Multi-task learning with code-text transformer. *arXiv preprint arXiv:2105.08645* (2021).
- [75] PHAN, L., TRAN, H., LE, D., NGUYEN, H., ANIBAL, J., PELTEKIAN, A., AND YE, Y. Cotext: Multi-task learning with code-text transformer, 2021.
- [76] PURBA, M. D., GHOSH, A., RADFORD, B. J., AND CHU, B. Software vulnerability detection using large language models. In *2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW)* (2023), pp. 112–119.
- [77] REIMERS, N., AND GUREVYCH, I. Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084* (2019).
- [78] REVEALISSUE. Revealissue. <https://github.com/VulDetProject/ReVeal/issues>, 2007. Accessed: 2023-07-29.
- [79] ROZIERE, B., GEHRING, J., GLOECKLE, F., SOOTLA, S., GAT, I., TAN, X. E., ADI, Y., LIU, J., REMEZ, T., RAPIN, J., ET AL. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).
- [80] RUSSELL, R., KIM, L., HAMILTON, L., LAZOVICH, T., HARER, J., OZDEMIR, O., ELLINGWOOD, P., AND MCCONLEY, M. Automated vulnerability detection in source code using deep representation learning. In *international conference on machine learning and applications* (2018), IEEE, pp. 757–762.
- [81] SARD. Nist software assurance reference dataset. <https://samate.nist.gov/SARD/>, 2004. Accessed: 2021-06-04.
- [82] SHEIKHALISHAHI, M., SARACINO, A., MEJRI, M., TAWBI, N., AND MARTINELLI, F. Fast and effective clustering of spam emails based on structural similarity. In *International Symposium on Foundations and Practice of Security* (2015), Springer, pp. 195–211.
- [83] SHESTOV, A., CHESHKOV, A., LEVICHEV, R., MUSSABAYEV, R., ZADOROZHNY, P., MASLOV, E., VADIM, C., AND BULYCHEV, E. Finetuning large language models for vulnerability detection. *arXiv preprint arXiv:2401.17010* (2024).
- [84] SHIMMI, S., AND RAHIMI, M. Leveraging code-test co-evolution patterns for automated test case recommendation. In *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test* (2022), pp. 65–76.
- [85] SHIMMI, S., AND RAHIMI, M. Mining software repositories for patternizing attack-and-defense co-evolution. In *Proceedings of the 1st International Workshop on Mining Software Repositories Applications for Privacy and Security* (2022), pp. 2–6.
- [86] SIMON, F., STEINBRUCKNER, F., AND LEWERENTZ, C. Metrics based refactoring. In *Proceedings fifth european conference on software maintenance and reengineering* (2001), IEEE, pp. 30–38.
- [87] SKOWYRA, R., CASTEEL, K., OKHRAVI, H., ZELDOVICH, N., AND STREILEIN, W. Systematic analysis of defenses against return-oriented programming. In *Research in Attacks, Intrusions, and Defenses* (2013), S. J. Stolfo, A. Stavrou, and C. V. Wright, Eds., Springer Berlin Heidelberg, pp. 82–102.
- [88] SKOWYRA, R., GOMEZ, S. R., BIGELOW, D., LANDRY, J., , AND OKHRAVI, H. QUASAR: Quantitative Attack Space Analysis and Reasoning. In *Proceedings of IEEE Annual Computer Security Applications Conference (ACSAC'17)* (Dec 2017).
- [89] SONG, D., LETTNER, J., RAJASEKARAN, P., NA, Y., VOLCKAERT, S., LARSEN, P., AND FRANZ, M. Sok: Sanitizing for security. In *2019 IEEE Symposium on Security and Privacy (SP)* (2019), IEEE, pp. 1275–1295.
- [90] SRIVASTAVA, P., PENG, H., LI, J., OKHRAVI, H., SHROBE, H., AND PAYER, M. FirmFuzz: Automated IoT Firmware Introspection and Analysis. In *Proceedings of the ACM CCS IoT Security & Privacy Workshop (IoTS&P)* (Nov 2019).
- [91] STEENHOEK, B., RAHMAN, M. M., JILES, R., AND LE, W. An empirical study of deep learning models for vulnerability detection. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)* (2023), IEEE, pp. 2237–2248.
- [92] STEENHOEK, B., RAHMAN, M. M., ROY, M. K., ALAM, M. S., BARR, E. T., AND LE, W. A comprehensive study of the capabilities of large language models for vulnerability detection. *arXiv preprint arXiv:2403.17218* (2024).
- [93] SUN, H., CUI, L., LI, L., DING, Z., HAO, Z., CUI, J., AND LIU, P. Vdsimilar: Vulnerability detection based on code similarity of vulnerabilities and patches. *Computers & Security* 110 (2021), 102417.
- [94] TANG, W., TANG, M., BAN, M., ZHAO, Z., AND FENG, M. Csgvd: A deep learning approach combining sequence and graph embedding for source code vulnerability detection. *Journal of Systems and Software* 199 (2023), 111623.
- [95] TANG, Z., HU, Q., HU, Y., KUANG, W., AND CHEN, J. Sevuldet: A semantics-enhanced learnable vulnerability detector. In *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2022), IEEE, pp. 150–162.
- [96] TAO, W., SU, X., WAN, J., WEI, H., AND ZHENG, W. Vulnerability detection through cross-modal feature enhancement and fusion. *Computers & Security* (2023), 103341.
- [97] TERRA, R., VALENTE, M. T., AND ANQUETIL, N. A lightweight modularization process based on structural similarity. In *2016 X Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS)* (2016), IEEE, pp. 111–120.
- [98] TIAN, J., XING, W., AND LI, Z. Bvdetect: A program slice-based binary code vulnerability intelligent detection system. *Information and Software Technology* 123 (2020), 106289.
- [99] TSANTALIS, N., AND CHATZIGEORGIOU, A. Identification of move method refactoring opportunities. *IEEE Transactions on Software Engineering* 35, 3 (2009), 347–367.
- [100] TSANTALIS, N., AND CHATZIGEORGIOU, A. Identification of extract method refactoring opportunities for the decomposition of methods. *Journal of Systems and Software* 84, 10 (2011), 1757–1782.
- [101] TURNER, S. Security vulnerabilities of the top ten programming languages: C, java, c++, objective-c, c#, php, visual basic, python, perl, and ruby. *Journal of Technology Research* 5 (2014), 1.
- [102] VELICKOVIC, P., CUCURULL, G., CASANOVA, A., ROMERO, A., LIO, P., BENGIO, Y., ET AL. Graph attention networks. *stat* 1050, 20 (2017), 10–48550.

- [103] VIDAS, T., LARSEN, P., OKHRAVI, H., AND SADEGHI, A.-R. Changing the game of software security. *Security Privacy, IEEE 16*, 2 (Mar/Apr 2018), 10–11.
- [104] WAN, Y., ZHAO, W., ZHANG, H., SUI, Y., XU, G., AND JIN, H. What do they capture? a structural analysis of pre-trained language models for source code. In *Proceedings of the 44th International Conference on Software Engineering* (2022), pp. 2377–2388.
- [105] WANG, H., YE, G., TANG, Z., TAN, S. H., HUANG, S., FANG, D., FENG, Y., BIAN, L., AND WANG, Z. Combining graph-based learning with automated data collection for code vulnerability detection. *IEEE Transactions on Information Forensics and Security 16* (2020), 1943–1958.
- [106] WANG, S., LIU, T., AND TAN, L. Automatically learning semantic features for defect prediction. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)* (2016), IEEE, pp. 297–308.
- [107] WANG, Y., LE, H., GOTMARE, A. D., BUI, N. D., LI, J., AND HOI, S. C. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922* (2023).
- [108] WANG, Y., WANG, W., JOTY, S., AND HOI, S. C. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* (2021).
- [109] WEN, X.-C., CHEN, Y., GAO, C., ZHANG, H., ZHANG, J. M., AND LIAO, Q. Vulnerability detection with graph simplification and enhanced graph representation learning. *arXiv preprint arXiv:2302.04675* (2023).
- [110] WU, P., YIN, L., DU, X., JIA, L., AND DONG, W. Graph-based vulnerability detection via extracting features from sliced code. In *2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C)* (2020), pp. 38–45.
- [111] YAMAGUCHI, F., WRESSNEGGER, C., GASCON, H., AND RIECK, K. Chucky: Exposing missing checks in source code for vulnerability discovery. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), pp. 499–510.
- [112] YING, A. T., MURPHY, G. C., NG, R., AND CHU-CARROLL, M. C. Predicting source code changes by mining change history. vol. 30, pp. 574–586.
- [113] YUAN, B., LU, Y., FANG, Y., WU, Y., ZOU, D., LI, Z., LI, Z., AND JIN, H. Enhancing deep learning-based vulnerability detection by building behavior graph model. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)* (2023), IEEE, pp. 2262–2274.
- [114] ZENG, P., LIN, G., PAN, L., TAI, Y., AND ZHANG, J. Software vulnerability analysis and discovery using deep learning techniques: A survey. *IEEE Access 8* (2020), 197158–197172.
- [115] ZHANG, C., LIU, B., XIN, Y., AND YAO, L. Cpvdt: Cross project vulnerability detection based on graph attention network and domain adaptation. *IEEE Transactions on Software Engineering* (2023).
- [116] ZHANG, H., BI, Y., GUO, H., SUN, W., AND LI, J. Isvsf: Intelligent vulnerability detection against java via sentence-level pattern exploring. *IEEE Systems Journal 16*, 1 (2021), 1032–1043.
- [117] ZHAO, Q., HUANG, C., AND DAI, L. Vuldeff: vulnerability detection method based on function fingerprints and code differences. *Knowledge-Based Systems 260* (2023), 110139.
- [118] ZHENG, W., ZHANG, M., TANG, H., CAI, Y., CHEN, X., WU, X., AND SEMASABA, A. O. A. Automatically identifying bug reports with tactical vulnerabilities by deep feature learning. In *International Symposium on Software Reliability Engineering* (2021), IEEE, pp. 333–344.
- [119] ZHOU, Y., LIU, S., SIOW, J., DU, X., AND LIU, Y. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *Advances in Neural Information Processing Systems* (2019), H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32, Curran Associates, Inc.
- [120] ZHOU, Y., AND SHARMA, A. Automated identification of security issues from commit messages and bug reports. In *Joint Meeting on Foundations of Software Engineering* (2017), Association for Computing Machinery, p. 914–919.
- [121] ZHU, Y., LIN, G., SONG, L., AND ZHANG, J. The application of neural network for software vulnerability detection: a review. *Neural Computing and Applications* (2022), 1–23.

A Appendix

6737 Original Code

CVE : CVE-2019-15938

CWE : CWE-119

```

static int nfs_readlink_req(struct nfs_priv *npriv, struct nfs_fh *fh,
                           char **target)
{
    uint32_t data[1024];
    uint32_t *p;
    uint32_t len;
    struct packet *nfs_packet;
    /*
     * struct READLINK3args {
     *   nfs_fh3 symlink;
     * };
     *
     * struct READLINK3resok {
     *   post_op_attr symlink_attributes;
     *   nfs_path3 data;
     * };
    */

```

This code makes an NFS (Network File System) READLINK request to retrieve the target of a symbolic link from an NFS server. The function prepares an NFS READLINK request to fetch the target of the symbolic link from the NFS server.

Mouse-over Info.
(details of neighbor's weakness)

Mouse-over Info.
(details of neighbor's purpose, through LLM)

*Mouse-click: The body of the methods (input and neighbor pairs) show in a separate window side by side when clicked

Contextual Dimension		CWE-254: Incorrect Permission Assignment for Critical Resource		The function retrieves the color value from an HTML input element
Neighbor 1 : 5837	similarity : 0.996	CVE : CVE-2015-1274	CWE : CWE-254	<code>colorValue()</code>
Neighbor 2 : 2323	similarity : 0.996	CVE : CVE-2013-1958	CWE : CWE-264	<code>int scm_check_creds(struct ucred *creds)</code>
Semantic Dimension		CWE-119: Improper Restriction of Operations within Bounds of a Memory Buffer		The function retrieves the color value from an HTML input element
Neighbor 1 : 2005	similarity : 0.749	CVE : CVE-2019-15937	CWE : CWE-119	<code>int nfs_readlink_reply(unsigned char *pkt, unsigned len)</code>
Neighbor 2 : 5071	similarity : 0.705	CVE : CVE-2017-12898	CWE : CWE-125	<code>interp_reply(netdissect_options *ndo, const struct s</code>
Syntactical Dimension		CWE-285: Improper Authorization		This code is part of a Linux kernel security check, related to privilege and capability handling. The code contains a set of nested conditions that check the privilege and capability of the calling process.
Neighbor 1 : 1344	similarity : 0.802	CVE : CVE-2016-7097	CWE : CWE-285	<code>int __gfs2_set_acl(struct inode *inode, struct posix_ac</code>
Neighbor 2 : 2452	similarity : 0.8	CVE : CVE-2015-4036	CWE : CWE-119	<code>vhos_t_scsi_make_tpg(struct se_wwn *wwn, struct config_</code>

Figure 4: An example report generated by VulSim containing the closest neighbors in each space.