Distinguishing Look-Alike Innocent and Vulnerable Code by Subtle Semantic Representation Learning and Explanation

Chao Ni
chaoni@zju.edu.cn
Zhejiang University
China

Xin Yin
xyin@zju.edu.cn
Zhejiang University
China

Kaiwen Yang
kwyang@zju.edu.cn
Zhejiang University
China

Dehai Zhao
dehai.zhao@anu.edu.au
Australian National University
Australia

Zhenchang Xing
zhenchang.xing@anu.edu.au
CSIRO's Data61&Australian National University
Australia

Xin Xia*
xin.xia@acm.org
Huawei
China

ABSTRACT

Though many deep learning (DL)-based vulnerability detection approaches have been proposed and indeed achieved remarkable performance, they still have limitations in the generalization as well as the practical usage. More precisely, existing DL-based approaches (1) perform negatively on prediction tasks among functions that are lexically similar but have contrary semantics; (2) provide no intuitive developer-oriented explanations to the detected results.

In this paper, we propose a novel approach named SVulD, a function-level Subtle semantic embedding for Vulnerability Detection along with intuitive explanations, to alleviate the above limitations. Specifically, SVulD firstly trains a model to learn distinguishing semantic representations of functions regardless of their lexical similarity. Then, for the detected vulnerable functions, SVulD provides natural language explanations (e.g., *root cause*) of results to help developers intuitively understand the vulnerabilities. To evaluate the effectiveness of SVulD, we conduct large-scale experiments on a widely used practical vulnerability dataset and compare it with four state-of-the-art (SOTA) approaches by considering five performance measures. The experimental results indicate that SVulD outperforms all SOTAs with a substantial improvement (i.e., 23.5%-68.0% in terms of F1-score, 15.9%-134.8% in terms of PR-AUC and 7.4%-64.4% in terms of Accuracy). Besides, we conduct a user-case study to evaluate the usefulness of SVulD for developers on understanding the vulnerable code and the participants' feedback demonstrates that SVulD is helpful for development practice.

CCS CONCEPTS

• **Security and privacy → Software security engineering**.

KEYWORDS

Vulnerability Detection, Developer-oriented Explanation, Subtle Semantic Difference, Contrastive Learning

ACM Reference Format:
Chao Ni, Xin Yin, Kaiwen Yang, Dehai Zhao, Zhenchang Xing, and Xin Xia. 2023. Distinguishing Look-Alike Innocent and Vulnerable Code by Subtle Semantic Representation Learning and Explanation. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3611643.3616358

1 INTRODUCTION

Software vulnerabilities have caused massive damage to software systems and many automatic vulnerability detection approaches have been proposed to prevent software systems from severity attacks and indeed achieved promising results, which can be broadly classified into two categories: static analysis approaches [1, 2, 17, 25, 26, 41] and deep learning (DL) approaches [7, 8, 11, 16, 28–31, 43–45]. The static analysis approaches focus on detecting type-specific vulnerabilities (i.e., user-after-free) with the help of user-defined rules or patterns, which highly depend on expert knowledge and have little chance to find a wider range of vulnerabilities [7, 12]. The deep learning approaches, benefiting from the powerful learning ability of deep neural networks, aim at leveraging advanced models to capture program semantics to identify potential type-agnostic software vulnerabilities. That is, these approaches automatically extract implicit vulnerability patterns from previous vulnerable code instead of requiring expert involvement, which makes deep learning become a good choice to solve vulnerability detection problems. However, the existing DL-based approaches still have two limitations that affect their effectiveness of generalization and the usefulness of development practice.

The first problem is that existing DL-based approaches have limited ability to distinguish subtle semantic differences among lexically similar functions. For a specific version of a vulnerable function, the vulnerabilities are usually fixed with a few modifications to it (i.e., 52.6% vulnerable functions can be fixed within 5 (76.7% within 10) lines of code in our dataset). The fixed functions can be conceptually treated as *non-vulnerable* functions. Meanwhile,

*Xin Xia is the corresponding author.

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA

ACM ISBN 979-8-4007-0327-0/23/12…$15.00
https://doi.org/10.1145/3611643.3616358

we find that *the vulnerable function and its corresponding fixed function are extremely lexically similar (i.e., fixing a vulnerability by modifying less than 100 CHARs accounts for 46.0% (200 for 65.1%)) but they have significant semantic differences (i.e., vulnerable or non-vulnerable)*. Ideally, we expect that a good-performing DL-based approach can perform equally well in detecting vulnerable functions and their corresponding fixing patches. However, we find that the SOTA DL-based approaches perform negatively on the fixed functions (i.e., non-vulnerable ones) and incorrectly classify the fixed version as vulnerable ones (43.5%-63.1% false positive). Thus, it is urgently required to pay more attention to semantic differences among lexically similar functions with contrasting semantics.

The second problem is that existing vulnerability detection approaches focus on giving binary detection results (i.e., vulnerable or not) and ignore the importance of providing developer-oriented natural-language explanations for the results. For example, *what is the possible root cause of such vulnerability? what impacts will be caused by this vulnerability?* Those explanations may help developers to have a better understanding of the detected vulnerable code. However, considering the concealment of software vulnerabilities, it is hard to observe two identical vulnerabilities. It is believed that similar/homogeneous vulnerabilities have similar root causes or lead to similar impacts. Intuitively, we find that many publicly available developer forums (i.e., Stack Overflow) share semantically similar problematic source code, and some of the responses provide useful and understandable natural language explanations about the issues, which help developers to intuitively figure out the potential root cause inside their problematic code.

To mitigate the above two limitations, we propose a novel approach named SVulD, which is a function-level Subtle semantic embedding for Vulnerability Detection along with intuitive explanations. It is technically based on pre-trained semantic embedding [22] as well as contrastive learning [10]. Specifically, to solve the first issue, SVulD adopts contrastive learning to train the UniXcoder [22] semantic embedding model in order to learn the semantic representation of functions regardless of their lexically similar information. To address the second issue, we build a knowledge-based crowdsource dataset by crawling problematic codes from Stack Overflow and fine-tune a BERT question-answering model [14, 39] on 1,678 manually labeled posts to automatically extract the key information from high-quality answers, which can provide developers with intuitive explanations and help them to understand the detected vulnerable code.

To evaluate the effectiveness of SVulD, we conduct extensive experiments on widely used practical vulnerability dataset [12, 27, 35]. Particularly, our SVulD is compared with four SOTA approaches (i.e., Devign, ReVeal, IVDetect, and LineVul) by five performance measures (i.e., Accuracy, Precision, Recall, F1-score, and PR-AUC). The experimental results indicate that SVulD outperforms all SOTA baselines with a substantial improvement (i.e., 23.5%-68.0% in terms of F1-score, 15.9%-134.8% in terms of PR-AUC and 7.4%-64.4% in terms of Accuracy). Besides, to provide developers with an intuitive explanation of the detected vulnerable code, we design a quality-first sorting strategy to prioritize the retrieved semantic-related post answers. We conduct a user-case study to evaluate whether our tool can help developers understand the problems in code intuitively and the participants' feedback demonstrates the usefulness of SVulD. Finally, this paper makes the main contributions as below:

Vulnerable Function

```
01 lookup_bytestring(netdissect_options *ndo, register const
   u_char *bs, const unsigned int nlen)
02 {
03   struct enamemem *tp;
     ......
16   while (tp->e_nxt)
17     if (tp->e_addr0 == i &&
18         tp->e_addr1 == j &&
19         tp->e_addr2 == k &&
20         memcmp((const char *)bs, (const char *)(tp-
           >e_bs), nlen) == 0)
           return tp;
       else
23         tp = tp->e_nxt;
24   tp->e_addr0 = i;
25   tp->e_addr1 = j;
26   tp->e_addr2 = k;
27   tp->e_bs = (u_char *) calloc(1, nlen + 1);
28   if (tp->e_bs == NULL)
       (*ndo->ndo_error)(ndo, "lookup_bytestring: calloc");
31   memcpy(tp->e_bs, bs, nlen);
32   tp->e_nxt = (struct enamemem *)calloc(1, sizeof(*tp));
33   if (tp->e_nxt == NULL)
       (*ndo->ndo_error)(ndo, "lookup_bytestring: calloc");
     return tp;
   }
```

Fixed/Clean Function

```
01 lookup_bytestring(netdissect_options *ndo, register const
   u_char *bs, const unsigned int nlen)
02 {
03   struct bsnamemem *tp;
     ......
16   while (tp->bs_nxt)
17     if (nlen == tp->bs_nbytes &&
18         tp->bs_addr0 == i &&
19         tp->bs_addr1 == j &&
20         tp->bs_addr2 == k &&
21         memcmp((const char *)bs, (const char *)
           (tp->bs_bytes), nlen) == 0)
           return tp;
       else
24         tp = tp->bs_nxt;
25   tp->bs_addr0 = i;
26   tp->bs_addr1 = j;
27   tp->bs_addr2 = k;
28   tp->bs_bytes = (u_char *) calloc(1, nlen + 1);
29   if (tp->bs_bytes == NULL)
       (*ndo->ndo_error)(ndo, "lookup_bytestring: calloc");
32   memcpy(tp->bs_bytes, bs, nlen);
33   tp->bs_nbytes = nlen;
34   tp->bs_nxt = (struct bsnamemem *)calloc(1,
     sizeof(*tp));
35   if (tp->bs_nxt == NULL)
       (*ndo->ndo_error)(ndo, "lookup_bytestring: calloc");
     return tp;
   }
```

Figure 1: An Out-of-bounds Read Vulnerability (CVE-2017-12894) in tcpdump

- We propose SVulD, a novel function-level approach for vulnerability detection with intuitive explanations based on the pre-trained semantic embedding model, which leverages contrastive learning technology to obtain the distinguishing semantic representations among lexically similar functions.
- We comprehensively investigate the effectiveness of SVulD on vulnerability detection and the generalization of fixed functions. The experiment results indicate that SVulD outperforms SOTAs with a substantial improvement (e.g., 23.5%-68.0% in terms of F1-score, 15.9%-134.8% in terms of PR-AUC). Especially, SVulD has better generalization performance on fixed functions (e.g., 7.4%-64.4% in terms of Accuracy).
- To the best of our knowledge, we are first to provide an intuitive explanation of the results given by a vulnerability detection approach, and a user-case study confirms the feasibility of intuitively explaining the results with crowdsourced knowledge.

2 MOTIVATING EXAMPLE

Functions usually consist of several lines of code for implementing a specific program semantic (i.e., functionality) and we use different labels (i.e., *vulnerable*, *non-vulnerable*) to describe the security status of functions. A vulnerable function includes security defects (e.g., CWE-125: Out-of-bounds Read) in its codes, while a non-vulnerable function is clean. A fixed function previously contains vulnerable codes but these codes have been fixed with some modifications on the vulnerable code snippets. Therefore, the fixed functions can be conceptually treated as *non-vulnerable* functions.

Fig. 1 shows two versions (the left one is for the vulnerable version, while the right one is for the non-vulnerable version) of a specific function in *tcpdump* project [21]. This function contains a typical *Out-of-bounds Read* vulnerability CVE-2017-12894. The *if condition statement* does not detect the length of the address

at line 17. *Comparing the left vulnerable one with the right non-vulnerable/fixed one, we find that the two versions are lexically similar but have distinguishing semantic differences from the security perspective*, which is not an accidental phenomenon. We conduct statistical analysis about the vulnerable functions as well as their corresponding fixed functions on the widely used dataset named Big-Vul (10,900 vulnerable functions) collected by Fan et al. [18] and find that 52.6% vulnerable functions can be fixed within five lines of codes (LOCs, added or deleted lines) and 76.7% functions can be fixed with less than 10 LOCs. From the view of modified chars, fixing a vulnerability by modifying less than 100 chars accounts for 46.0% (200 chars for 65.1%). Meanwhile, a function has a ratio of no more than 5% accounts for 48.7% (10% for 63.4%) between the number of modified chars and the whole number of chars. All these statistical results indicate that the vulnerable function and the corresponding fixed function are extremely lexically similar.

Recently, benefiting from the powerful learning ability of deep neural networks, many SOTA DL-based vulnerability detection approaches (e.g., Devign [45], ReVeal [8], IVDetect [27], and LineVul [19]) have been proposed to capture program semantics in order to identify potential software vulnerabilities, and these approaches have achieved promising performance. Ideally, a good-performing DL-based approach is expected to have a good generalization ability, which means that the approach should work well on both vulnerable and corresponding fixed non-vulnerable functions. However, a large-scale experiment on Big-Vul shows that all these SOTA approaches have negative performance on predicting the fixed functions (i.e., non-vulnerable ones). Specifically, they incorrectly classify the fixed functions as vulnerable ones (43.5%-63.1% false positive, cf. Section 5.1 for details).

Meanwhile, almost all existing vulnerability detection approaches focus on classifying whether a function is vulnerable but do not provide developer-oriented natural-language explanations to help developers understand the detected vulnerable code. For example, *what is the possible root cause of such vulnerability? what impacts will be caused by this vulnerability?* Such types of explanations may (at least intuitively) help developers to have a deeper understanding of the detected vulnerable code. Intuitively, many publicly available user forums (i.e., Stack Overflow) share similar problematic source code and their corresponding responses may provide useful and understandable natural language explanations about the issues, which can intuitively help developers to figure out the potential root cause inside the vulnerable code.

As shown in Fig. 2, this code snippet has a similar root cause with the vulnerable function in Fig. 1. It crashes because of the limited size of defined arrays (i.e., *teams* and *wonGames*), which results in an *Out-of-bounds* error when reading and writing content to the last element. Similarly, the function in Fig. 1 will crash when the last element in their address array does not satisfy the length of a legal internet address. If developers are provided with a natural language explanation of the root cause referring to the answer in Fig. 2, the problem in Fig. 1 will be easier to solve.

Motivating. *Two code snippets may be lexically similar but have distinct security semantics (vulnerable or non-vulnerable), which needs to embed their semantic difference in a better way. Meanwhile, similar vulnerabilities may have a similar root cause, which can help participants understand the problematic codes better.*

Figure 2: A simple but similar problematic code along with an accepted answer in Stack Overflow.

3 OUR APPROACH: SVULD

To investigate the feasibility of our intuitive hypothesis, we propose a novel framework named SVulD, which integrates software vulnerability detection and intuitive natural language explanation. As illustrated in Fig. 3, SVulD consists of two main phases: ❶ training phase, where the vulnerability detector is trained on the high-quality dataset and vulnerability explainer is constructed on crowd-sourced knowledge; ❷ inference phase, where a specific function is classified as vulnerable or not by the trained vulnerability detector and provide several developer-oriented explanations to the detected vulnerable function. We present the details of SVulD in the following subsections.

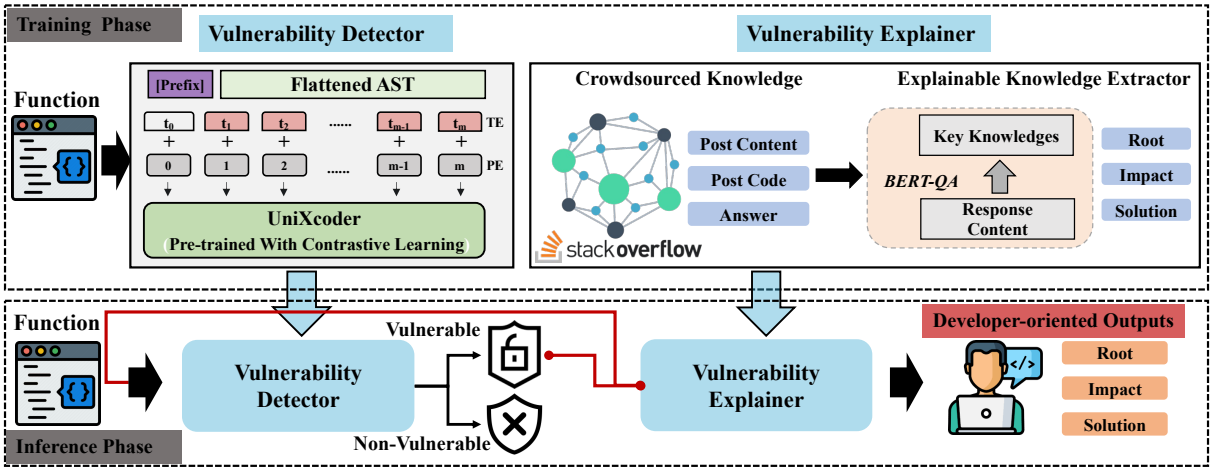


Figure 3: The framework of SVulD.

3.1 Vulnerability Detection

In order to discriminate the semantic difference among lexically similar functions effectively, SVulD adopts contrastive learning framework with the pre-trained model, UniXcoder [22], as the semantic encoder. The architecture for contrastively training the UniXcoder-based semantic embedding model is illustrated in Fig. 4. Contrastive learning [36] is a kind of deep neural network training process that takes paired functions as input and uses the similarity between the paired functions as labels. The training objective of contrastive learning is to learn whether two functions are semantically similar regardless of their lexical similarity. Elaborately, the contrastive learning framework utilizes the encoder to embed source code into their semantic representations (i.e., hidden vectors) and aims at minimizing the distance between similar functions while maximizing the distance between dissimilar functions. There are two important components of the proposed model: an encoder for embedding functions' semantics and a learning strategy for discriminating differences.

3.1.1 Semantic Encoder. Considering many successful applications of pre-trained models in software engineering (e.g., defect prediction [33] and code summarization [46]), especially the recent work on vulnerability detection [19], we leverage UniXcoder [22] as our semantic encoder. It is a unified cross-modal (i.e., code, comment and abstract syntax tree (AST)) pre-trained model for programming language and utilizes mask attention matrices with prefix adapters (i.e., [*prefix*]) to control the behavior of the model (i.e., encoder only ([*Enc*]), decoder only ([*Dec*]) or encoder-decoder ([*E2D*])). For each input function, UniXcoder encodes the AST of it into a sequence while retaining all structural information of the tree. Meanwhile, in our binary classification setting, we set [*prefix*] as [*Enc*] and fine-tune it on our studied datasets to learn a better representation of source codes' semantic information.

3.1.2 Semantic Difference Learning. Our goal is to discriminate the semantic difference among lexical similar functions, which is consistent with the target of contrastive learning. That is, minimize the distance between similar objects (i.e., the function in our study) while maximizing the distance between dissimilar objects. Hoffer et al. [24] proposed the triplet network for contrastive learning, which requires a triplet (F, P, N) as the input, where F corresponds to the original source code of the function, P refers to the positive equivalent of F, and N is the negative one. In our work, for a given function F in the training data, its positive functions are the varying representation of the same functions and the negative functions are functions that are different from the given one. Therefore, with a good semantic presentation, similar functions stay close to each other while dissimilar ones are far apart.

Fig. 4 shows the architecture of the contrastive learning used in this work, in which the UniXcoder is the base model for semantic embedding. We use a Pooling layer to connect the UniXcoder model and the triple network. The triple network has two layers. The first layer is three identical deep neural networks for feature extraction of input functions, which can be easily replaced with other semantic learning models. The second layer of the triplet network is a loss function based on the cosine distance operator with transformation operations of *projector*, which is used to minimize the distance between similar functions and maximize the distance between dissimilar functions. The training objective is to fine-tune the network so that the distance between the functions F and the positive functions P is closer than the distance between the functions F and the negative functions N, which is illustrated below:

$$max(||E_F - E_P|| - ||E_F - E_N|| + \epsilon, 0) \quad (1)$$

where E_F, E_P, and E_N are the semantic embeddings of function S, P, and N respectively. ϵ is the margin of the distance between S and N. By default, ϵ is set to 1, which means the cosine distance between a function and its irrelevant function should be 1.

3.2 Vulnerability Explanation

Vulnerability explanation aims to provide developer-oriented natural language descriptions for problematic source code, which involves two aspects: building a code-related crowdsourced knowledge database and extracting key aspects for understanding vulnerable functions.

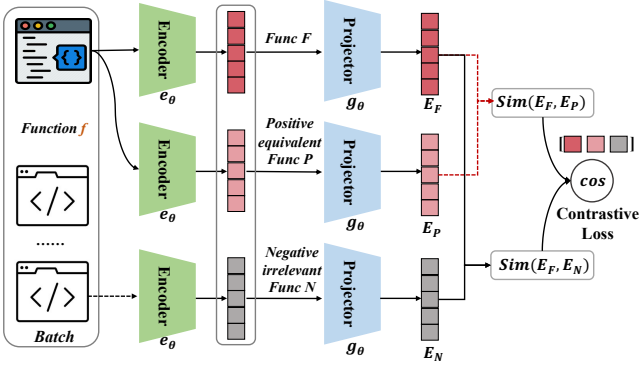


Figure 4: Architecture for contrastively training UniXcoder based semantic embedding model

3.2.1 Crowdsourced Knowledge Database. This phase aims at managing diverse and useful information from developer forums (i.e., Stack Overflow) since the developer forums provide a lot of information in the form of question and answer (Q&A) about (usually problematic) codes. Meanwhile, users can also vote on the answers to distinguish the value of the questions and the corresponding answers.

In our knowledge database, we focus on two objects: questions/posts about a technical problem and answers for solving this problem. For a question/post, it usually contains a title for concisely describing a problem, the details of the question, the source codes involved as well as an optional tag. For an answer, it has a label to indicate whether it is a suggested one. Meanwhile, the answer may give a detailed description about *why it arises the problem, where the root cause exists, and how to solve it, especially for the suggested one.* The descriptions are usually presented in the form of natural language, while the code presents the potential correctness solutions. The solution does not always work successfully for each user who is facing a similar problem because of environmental differences. However, an explanation of problematic codes will inspire other users who encounter similar problems to understand the root cause.

Additionally, we connect posts with the same tags for retrieving answers more efficiently in the next phase (i.e., Results Explainer), as this process can fuse related posts with relevant problems.

3.2.2 Result Explainer. The crowdsourced code knowledge database helps to fuse useful information when addressing similar problems, while the result explainer aims at both retrieving relevant questions/posts involving similar source codes and extracting key aspects of the problems from the suggested answers.

The first step is to figure out the most (especially semantically) relevant source codes explicitly. In this paper, for retrieving the most semantically similar problematic functions, we adopt UniXcoder to obtain semantic embedding of functions since the model has been well pre-trained with contrastive learning technology. Additionally, for a given retrieved post, there usually exists many responses from different users with varying experiences. All the diverse responses can be useful since different developers may give their responses in different development environments (i.e., issues that occurred in Windows OS or Linux OS). Therefore, apart from retrieving similar problematic functions, we also design an effective quality-first sorting strategy as follows to prioritize the most useful response/explanation.

$$Ranking\ Score = Func\_Sim \times \left(\frac{score_i}{\sum_{j=1}^{N} score_j} \times Asps.\right)$$
$$Asps. = 0.5 \times I(Cau.) + 0.3 \times I(Imp.) + 0.1 \times I(Sol.) + 0.1 \times I(Acpt.) \quad (2)$$

where $Func\_Sim$ represents the similarity between the code in a given post and the vulnerable function, $score_i$ means the score of an answer i in a post, which is voted by users. A high score usually reflects the high quality of the answer. N represents the number of answers in the given post and $I(\cdot)$ is an indicator function. It equals 1 if the condition is satisfied else it equals 0. For example, $I(Cau.)$ equals 1 when the answer contains the root cause description to explain a problem. In addition, it is possible that the root cause ($Cau.$), the impact ($Imp.$), and the potential solution ($Sol.$) provide different information for developers to understand the problems in codes. Therefore, we assign different weights to indicate their priority. Finally, if an answer is marked as *Accept* ($Acpt.$), it means the answer has high quality for solving the problem, and we take it into consideration and assign the weight to 0.1.

The second step is to extract the key aspects for understanding the problem. As introduced in the crowdsourced knowledge database, the suggested answer may contain a detailed description that explains key aspects (e.g., root cause, impact, solution, etc.) of the problem in source codes. In our explanation model, we focus on the following three aspects: *root cause*, *impact*, and *solution*, which are usually long clauses or sentences.

To extract the root cause, impact, and solution, we leverage the BERT-based Question Answering model [14, 39], which is based on a pre-trained BERT model for retrieving questions and answers in a given content scope. The input of the model includes a question and the scope for answering the question. The model outputs the start and end word index as the answer clause. In our application of BERT-QA, we adopt the *content* of a suggested answer as the scope of question answering, and we input three what-is questions (i.e., "*what is root cause*", "*what is impact*" and "*what is the solution*") into the model to find corresponding answers. Benefiting from the language modeling capability of BERT, BERT-QA can handle complex clauses of the root cause, impact, and solution, and select the most appropriate information from the long response texts.

We train the BERT-QA model with 1,678 question-answer pairs (920 of reasons, 391 of impacts, and 492 of solutions), which is constructed manually from 55,627 posts (121,635 answers) in Stack Overflow. We build both positive and negative questions for which the answers can or cannot be found in the given posts. The negative questions help the model to learn when it fails to find any answer in the scope. This characteristic is extremely important for extracting the root cause, impact, and solution since not all posts exactly and completely describe all three aspects. Otherwise, the BERT-QA model will have no ability to handle negative questions and extract some irrelevant content as the answer for a question.

4 EXPERIMENTAL DESIGN

In this section, we first present features of the studied datasets, and then introduce the baseline approaches. Following that, we describe the performance metrics as well as the experimental settings.

4.1 Datasets

Vulnerability Dataset. We use the benchmark dataset provided by Fan et al. [18] due to the following reasons. The first one is to establish a fair comparison with existing approaches (e.g., IVDetect, LineVul). The second one is to evaluate whether existing approaches have a good generalization performance on detecting the fixed functions since Fan et al. [18]'s dataset is the only one vulnerability dataset that provides the fixed version of vulnerable functions. The last one is to satisfy the distinct characteristics of the real world as well as the diversity in the dataset, which is suggested by previous works [8, 23].

Fan et al. [18] built the large-scale C/C++ vulnerability dataset named Big-Vul from Common Vulnerabilities and Exposures (CVE) database and open-source projects. Big-Vul totally contains 3,754 code vulnerabilities collected from 348 open-source projects spanning 91 different vulnerability types from 2002 to 2019. It has 188,636 C/C++ functions with a vulnerable ratio of 5.7% (i.e., 10,900 vulnerability functions). The authors linked the code changes with CVEs as well as their descriptive information to enable a deeper analysis of the vulnerabilities. In our work, some baselines need to obtain the structure information (e.g., control flow graph (CFG), data flow graph (DFG)) of the studied functions. Therefore, we adopt the same toolkit with *Joern* [4] to transform functions. The functions are dropped out directly if they cannot be transformed by *Joern* successfully. We also remove the duplicated functions and the statistics of the studied dataset are shown in Table 1.

Table 1: The statistic of studied dataset

Datasets	# Vul.	# Non-Vul.	# Total	% Vul.: Non-Vul.
Original Big-Vul	10,900	177,736	188,636	0.061
Filtered Big-Vul	5,260	96,308	101,568	0.055
Training	4,208	4,208	8,416	1
Validating	526	9,631	10,157	0.055
Testing	526	9,631	10,157	0.055

Crowdsourced Dataset. Apart from the widely used vulnerability dataset, we also need to build a crowdsourced dataset manually in order to provide explanations for the detected vulnerabilities. In this paper, we crawl posts as well as their answers from Stack Overflow, where the posts are labeled with C or C++ and there is at least one code snippet in their content. Finally, we obtain 55,627 posts with 121,635 answers, which are further used to build a knowledge database.

4.2 Baselines

To comprehensively compare the performance of SVulD with existing work, in this paper, we consider the four SOTA approaches: Devign [45], ReVeal [8], IVDetect [27], and LineVul [19]. We briefly introduce them as follows.

Devign proposed by Zhou et al. [45] is a general graph neural network based model for graph-level classification through learning on a rich set of code semantic representations including AST, CFG, DFG, and code sequences. It uses a novel *Conv* module to efficiently extract useful features in the learned rich node representations for graph-level classification.

ReVeal proposed by Chakraborty et al. [8] contains two main phases: feature extraction and training. In the former phase, ReVeal translates code into a graph embedding, and in the latter phase, ReVeal trains a representation learner on the extracted features to obtain a model that can distinguish the vulnerable functions from non-vulnerable ones.

IVDetect proposed by Li et al. [27] involves two components: coarse-grained vulnerability detection and fine-grained interpretation. As for vulnerability detection, they process the vulnerable code and the surrounding contextual code in a function distinctively, which can help to discriminate the vulnerable code and the benign ones. In particular, IVDetect represents source code in the form of a program dependence graph (PDG) and treats the vulnerability detection problem as graph-based classification via graph convolution network with feature attention. As for interpretation, IVDetect adopts a GNNExplainer to provide fine-grained interpretations that include the sub-graph in PDG with crucial statements that are relevant to the detected vulnerability.

LineVul proposed by Fu et al. [19] is a Transformer-based line-level vulnerability prediction approach. LineVul leverages BERT architecture with self-attention layers which can capture long-term dependencies within a long sequence. Besides, benefiting from the large-scale pre-trained model, LineVul can intrinsically capture more lexical and logical semantics for the given code input. Moreover, LineVul adopts the attention mechanism of BERT architecture to locate the vulnerable lines for finer-grained detection.

4.3 Evaluation Measures

To evaluate the effectiveness of SVulD on vulnerability detection, we consider the following five metrics: Accuracy, Precision, Recall, F1-score, and PR-AUC.

Accuracy evaluates the performance that how many functions can be correctly labeled. It is calculated as: $\frac{TP+TN}{TP+FP+TN+FN}$.

Precision is the fraction of true vulnerabilities among the detected ones. It is defined as: $\frac{TP}{TP+FP}$.

Recall measures how many vulnerabilities can be correctly detected. It is defined as: $\frac{TP}{TP+FN}$.

F1-score is a harmonic mean of *Precision* and *Recall* and can be calculated as: $\frac{2\times P\times R}{P+R}$.

PR-AUC is the area under the precision-recall curve and is a useful metric of successful prediction when the class distribution is very imbalanced [23]. The precision-recall curve shows the trade-off between precision and recall for different thresholds. A high area under the curve indicates both high recall and high precision, where high precision corresponds to a low false positive rate, and high recall corresponds to a low false negative rate.

4.4 Experimental Setting

We implement our vulnerability detection and explanation model SVulD in Python with the help of PyTorch framework. Besides, we utilize *unixcoder-base-nine* [22] from Huggingface [3] as our basic model, which is a pre-trained model on NL-PL pairs of CodeSearchNet dataset and additional 1.5M NL-PL pairs of C, C++, and C# programming language. We fine-tune SVulD on the studied datasets to obtain a set of suitable parameters for the vulnerability

detection task and fine-tune BERT-QA model on the manually labeled question-answer datasets. All the models are fine-tuned on four NVIDIA GeForce RTX 3090 graphic cards. During the training phase, we use *Adam* with a batch size of 32 to optimize the parameters of SVulD. We also leverage *GELU* as the activation function. A dropout of 0.1 is used for dense layers before calculating the final probability. We set the maximum number of epochs in our experiment as 20 and adopt an early stop mechanism to obtain good parameters. The models (i.e., SVulD and baselines) with the best performance on the validation set are used for the evaluations.

5 EXPERIMENTAL RESULTS

To investigate the feasibility of SVulD on software vulnerability detection and detection result explanation, our experiments focus on the following four research questions:

- **RQ-1**. *To what extent can the function-level vulnerability detection performance SVulD achieve?*
- **RQ-2**. *How does the paired instance building strategy impact the performance of SVulD?*
- **RQ-3**. *How does the size of paired instance impact the performance of SVulD?*
- **RQ-4**. *How well does SVulD perform on explaining the detection results?*

In RQ1, we aim to investigate the performance of the SVulD on vulnerability detection by considering it with SOTA baselines (cf. Section 5.1). In RQ2 and RQ3, we explore the impact of design options of contrastive learning on the performance of SVulD (cf. Section 5.2, 5.3). In RQ4, we explore the SVulD's usefulness for helping developers understand vulnerable functions (cf. Section 5.4).

5.1 [RQ-1]: Effectiveness on Vulnerability Detection.

Objective. Benefiting from the powerful representation capability of deep neural networks, many DL-based vulnerability detection approaches have been proposed [27, 45]. However, as vulnerable functions are usually fixed with a few modifications (52.6% vulnerable functions can be fixed within 5 (76.7% for 10) lines of codes), they have subtle lexical differences with the non-vulnerable functions. Existing SOTA deep learning approaches (i.e., Devign, ReVeal, IVDetect, etc.) cannot perform well on the fixed functions (non-vulnerable). The main reason falls into the limitations of effective semantic embedding among lexical similar functions. In this paper, we propose a novel approach SVulD, which is built on a contrastive learning framework with a pre-trained model as a semantic encoder as suggested by previous work [9]. The experiments are conducted to investigate whether SVulD outperforms SOTA function-level vulnerability detection approaches.

Experimental Design. We consider the four SOTA baselines: Devign [45], ReVeal [8], IVDetect [27], and LineVul [19]. These approaches can be divided into two categories: GNN-based one (i.e., Devign, ReVeal and IVDetect) and Pre-trained-based one (i.e., LineVul). Besides, in order to comprehensively compare the performance among baselines and SVulD, we consider five widely used performance measures and conduct experiments on the popular dataset. Since GNN-based approaches usually need to obtain the structure information of the function (e.g., CFG, DFG), we adopt the same toolkit with *Joern* to transform functions. Finally, the filtered dataset (shown in Table 1) is used for evaluation. We follow the same strategy to build the training data, validating data, and testing data from the original dataset with previous work does [19, 34]. Specifically, 80% of functions are treated as training data, 10% of functions are treated as validation data, and the left 10% of functions are treated as testing data. We also keep the distribution as same as the original ones in training, validating, and testing data.

Meanwhile, for a specific function, SVulD needs to select appropriate positive instances and negative instances. For the positive instances, we adopt the different embedding vectors of the same function by randomly dropping out some weights in the network of the semantic encoder. For the negative instances, we consider all the other instances (i.e., functions) in the same mini-batch with the given instance and use the average semantic vector representation. We consider three types of paired instances selection strategies (i.e., *SimCL*, *SimDFE* and *R-Drop*. cf. Section 5.2), and in this RQ, we adopt the *R-Drop* strategy since it has overall best performance.

Finally, since our target is to build an effective vulnerability detection model, especially for discriminating lexically similar but semantically distinct functions, we further conduct an analysis on how SVulD performs on the fixed version of vulnerable functions in the testing dataset.

Table 2: Vulnerability detection results of SVulD compared against four baselines.

Methods	F1-score	Recall	Precision	PR-AUC
Devign	0.200	**0.660**	0.118	0.115
ReVeal	0.232	0.354	0.172	0.145
IVDetect	0.231	0.540	0.148	0.177
LineVul	0.272	0.620	0.174	0.233
SVulD	**0.336**	0.414	**0.282**	**0.270**
Improv.	23.5%-68.0%	–	62.1%-139.0%	15.9%-134.8%

Results. The evaluation results are reported in Table 2 and the best performances are highlighted in bold. According to the results, we find that our approach SVulD outperforms all SOTA baseline methods on almost all performance measures except *Recall*. In particular, SVulD obtains 0.336, 0.282, and 0.270 in terms of F1-score, Precision, and PR-AUC, which improves baselines by 23.5%-68.0%, 62.1%-139.0%, and 15.9%-134.8% in terms of F1-score, Precision, and PR-AUC, respectively.

In terms of *Recall*, Devign performs the best (0.660) and LineVul performs similarly with Devign (0.620), which means that both the pre-trained model and the GNN-based model can achieve better performance of *Recall*.

The performance comparisons of SVulD and four SOTAs on the fixed functions are presented in Table 3. According to Table 3, we find that all SOTAs have poor performance on classifying the fixed versions (i.e., the clean version) in the testing dataset (i.e., 526 vulnerable functions), while SVulD can achieve the best performance. More precisely, SVulD can correctly classify 319 fixed versions of functions as clean ones, which outperforms Devign (i.e., 194), ReVeal (i.e., 297), IVDetect (i.e., 209), and LineVul (i.e., 202) by 64.4%, 7.4%, 52.6%, and 57.9%, respectively. The results indicate

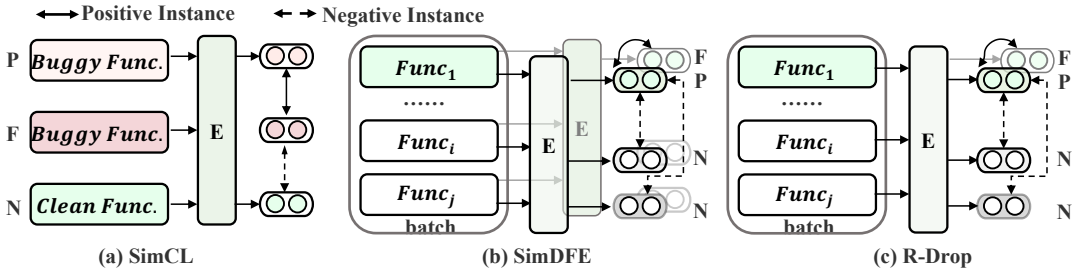


Figure 5: Three different contrastive paired instances construction

Table 3: The effectiveness of SVulD compared against four baselines on fixed functions in testing dataset

Methods	# Correct	Accuracy	# Improv.	% Improv.
Devign	194	36.9%	125	64.4%
ReVeal	297	56.5%	22	7.4%
IVDetect	209	39.7%	110	52.6%
LineVul	202	38.4%	117	57.9%
SVulD	**319**	**60.6%**	**22-125**	**7.4% - 64.4%**

that SVulD has a better representation of learning ability than the four baselines.

Answer to RQ-1: *SVulD outperforms the SOTA baselines at the function-level software vulnerability detection. Particularly, it achieves overwhelming results at both F1-score and PR-AUC, which indicates that SVulD equipped with contrastive learning as well as pre-trained model has a stronger ability to learn the semantics of functions, especially for those functions with lexical similarity but have distinct semantics.*

5.2 [RQ-2]: Impacts of Contrastive Paired Instances Construction.

Objective. The contrastive learning framework needs to build triplet paired instances, which are used to measure how close the two similar instances are and how far the two dissimilar instances are. Therefore, it is important to conduct a study on how the constructed positive instances and negative instances of a given function affect the learning of semantic representation.

Experimental Design. We consider three types (i.e., *SimCL*, *SimDFE*, and *R-Drop*) of paired instances (i.e., positive instances and negative instances) building strategies to train our proposed approach SVulD. The differences among these strategies are illustrated in Fig. 5 and we introduce them in detail as follows.

• **SimCL** (simple contrastive learning) means building the negative instance of a vulnerable function with its corresponding fixed version. For its positive equivalent function, we input the original function twice into the same encoder with different weights (i.e., dropout used as noise) inside the model and obtain two embedded vectors. The two vectors are interchangeably treated as positive instances.

• **SimDFE** (simple duplicate function embedding) means inputting all functions (noted as $f_1, f_2, \cdots, f_n$, n is the size of batch) in a batch twice into the same encoder with different weights, which is inspired by [20]. That is, each function will have two embedded vectors, noted as $f_{11}, f_{12}, f_{21}, f_{22}, \cdots, f_{n1}, f_{n2}$. Take f_1 as an example, f_{11} and f_{12} are interchangeably treated as positive instances and f_{ij} are treated as negative instances, where $i \in [2, n]$ and $j \in [1, 2]$. We use the average difference between f_1 and all negative instances as their dissimilarity.

• **R-Drop** (random dropout) means to input one function (noted as $f_1, f_2, \cdots, f_n$, n is the size of batch) in a batch twice and the rest function in the same batch once into the same encoder. For the given function embedded with an encoder twice, we adopt the random *dropout* operation to the network to obtain the equivalent positive embedding. Take f_1 as an example, f_{11} and f_{12} are interchangeably treated as positive instances and $f_i (i \in [2, n])$ are treated as negative instances. We use the average difference between f_1 with all negative instances as their dissimilarity.

The experimental dataset is set the same as the experiment of RQ-1 (i.e., 80% for training, 10% for validating, and 10% for testing). We also consider the five performance measures (i.e., Precision, Recall, F1-score, PR-AUC, and Accuracy) for comprehensively studying the impact of different paired instances building strategies. Additionally, in this study, we set the batch size n as 32.

Table 4: The performance difference among three different paired instances construction strategies

Strategy	Testing Data				Fixed function	
	F1-score	Recall	Precision	PR-AUC	# Num	Accuracy
SVulD	*0.303*	***0.536***	*0.211*	*0.245*	*243*	*0.462*
SVulD $_{SimCL}$	0.313	0.504	0.227	0.257	269	0.511
SVulD $_{SimDFE}$	0.324	0.481	0.244	0.265	268	0.510
SVulD $_{R-Drop}$	**0.336**	0.414	**0.282**	**0.270**	**319**	**0.606**
Improv.	3.3% to 10.9%	—	7.6% to 33.6%	4.9% to 10.2%	10.3% to 31.3%	

Results. The comparison results are reported in Table 4 and the best performances are highlighted in bold for each performance measure. According to the results, we can obtain the following observations: (1) All paired instance construction strategies have the advantage of learning function semantic embedding in the scenario of vulnerability detection. Particularly, *SimCL*, *SimDFE*, and *R-Drop* improve the baseline (UniXcoder without contrastive learning) by 3.3%-10.9%, 7.6%-33.6%, 4.9%-10.2%, and 10.3%-31.3% in terms of F1-score, Precision, PR-AUC, and Accuracy. (2) The *SimDFE* performs better than the *SimCL* and the *R-Drop* is the dominated one among the three strategies. (3) The *SimCL* performs

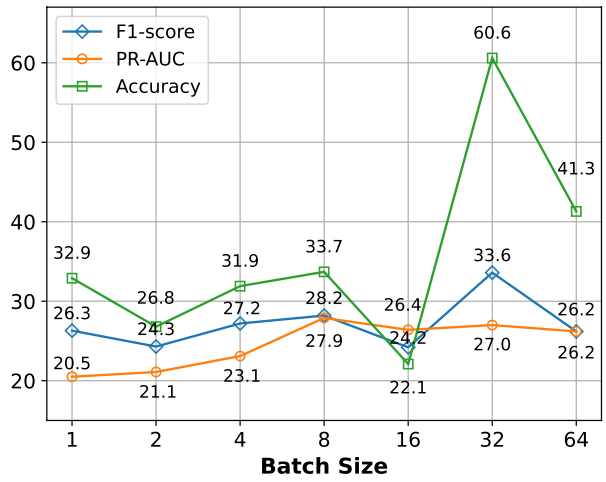


Figure 6: The varying performance of SVulD with different batch size

worse than the other two strategies and the main reason may come from the small size of negative instances (i.e., only one negative instance), which limits the information for SVulD to discriminate the difference between positive instances and negative instances. (4) The contrastive learning strategy, to some degree, can decrease the performance of *Recall*. However, it has an improvement on two comprehensive performance measures (i.e., *F1-score* and *PR-AUC*), especially for distinguishing two lexically similar functions with distinct semantics (i.e., an improvement on *Accuracy*).

Answer to RQ-2: *All paired instance construction strategies present their own advantages in learning function semantic embedding and the R-Drop strategy performs the best.*

5.3 [RQ-3]: Impacts of Paired Instances Size.

Objective. In RQ-2, we find that the number of negative instances has an impact on SVulD's performance of learning semantic embedding. Therefore, we want to conduct a deeper experiment on how the batch size (i.e., the number of negative instances) impacts the performance of SVulD on discriminating dissimilar instances from similar ones.

Experimental Design. According to RQ-2, we find that the *R-Drop* strategy has an overall better performance than the others. Meanwhile, considering the fact that the larger the batch size is, the more memory SVulD consumes, we re-run SVulD with *R-Drop* strategy on the following varying settings of batch size: 1, 2, 4, 8, 16, 32, and 64. Because of the limitation of graph memory (i.e., four NVIDIA RTX 3090) and the size of functions, we cannot perform larger batch sizes (i.e., 128 or 256). Besides, the experimental dataset is set as same as that in previous RQs. We evaluate the performance of SVulD on testing data with two comprehensive performance measures (i.e., F1-score and PR-AUC), and we adopt Accuracy to evaluate the performance on the fixed version of vulnerable functions.

Results. The evaluation results of SVulD with varying batch size are illustrated in Fig. 6. According to the results, we have the following research findings: (1) Different number of negative instance has varying impact on SVulD's performance. (2) Almost all the metrics of SVulD (except Accuracy) go up with the increasing of negative instances when batch size is no larger than 32. When batch size equals 64, all the performances drop to different degrees. (3) Larger batch size may not lead to better performance and assigning a batch size of 32 is a good choice.

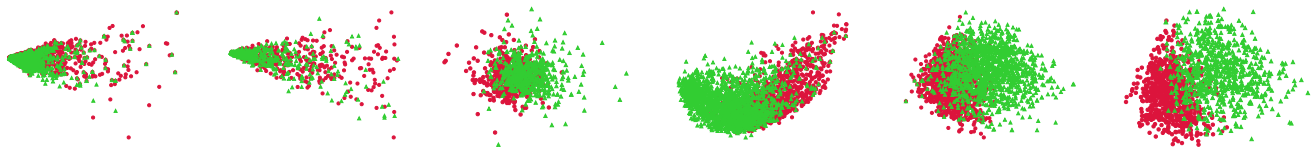
Answer to RQ-3: *The number of negative instances has an impact on SVulD's performance and the larger number may not always guarantee better performance. In our setting, a median size (i.e., 32) is more appropriate.*

5.4 [RQ-4]: Usefulness for Developers.

Objective. Though many novel approaches have been proposed and indeed achieved remarkable performance, existing methods cannot provide a developer-oriented, natural language-described explanation. For example, *what is the possible root cause of such vulnerability?* Such types of explanations may (at least intuitively) help developers understand the identified vulnerability better. However, considering the concealment of software vulnerabilities, we cannot observe two identical vulnerabilities. It is possible that similar/homogeneous vulnerabilities have similar root causes or lead to similar impacts. Meanwhile, many publicly available developer forums (i.e., Stack Overflow) share similar problems and their responses may provide understandable natural language explanations about the issues. Therefore, we want to further utilize this useful and diverse information to provide participants with detailed explanations about the identified problematic codes.

Experimental Design. We first crawl posts labeled with C/C++ from Stack Overflow and build a database (cf. Section 4.1) to fuse all crowdsourced knowledge for retrieving important explainable information. Considering that our work focuses on code-related problems, we filter those posts with no code snippet in their post content since the code snippet is the critical connective element when retrieving similar problematic codes. In addition, for retrieving the most semantically similar problematic functions, we adopt SVulD to obtain semantic embedding of both vulnerable function and code snippet in post since our model has been well pre-trained with contrastive learning technology. Then, we adopt the designed quality-first sorting strategy (cf. Section 3.2) to prioritize the retrieved answers. Finally, the well pre-trained BERT-QA model (cf. Section 3.2) is adopted to extract three optional important descriptions (i.e., *root cause*, *impact*, and *solution*) inside the answer.

Finally, we randomly select 20 vulnerable functions in testing datasets and invite 10 developers from a prominent IT company who have 5 to 8 years of experience in software security as our participants. Each developer is asked to finish an experiment task that includes two vulnerable functions as well as their corresponding explanation recommended by SVulD. We evaluate the usefulness of our approach by analyzing the answers to the following questions given by participants. More precisely, SVulD presents each vulnerable function with five retrieved answers from crowdsourced knowledge.



(a) Devign (b) ReVeal (c) IVDetect (d) LineVul (e) UniXcoder (f) SVulD

Figure 7: Visualization of the separation between vulnerable (denoted by ●) and non-vulnerable (denoted by ▲).

- Q1: Is the explanation related to the vulnerable function?
- Q2: Is the explanation comprehensive (i.e., the root cause, the impacts, and the suggestion. score: 1-5', 1(low)-3(middle)-5(high))? Which part is most important?
- Q3: Is the explanation useful to understand the vulnerability?
- Q4: In which result do you find the most desired answer? (score: 0-5', 0 means no desired answer.)
- Q5: Please sort the explanations according to their usefulness.

For Q1 and Q2, we aim to verify the relatedness and comprehensiveness of SVulD's recommendation. For Q3 and Q4, we aim to evaluate the usefulness of SVulD, and Q5 is designed to evaluate the difference between the recommendations and developers' expectations.

Results. In Q1, except for 5 negative responses (i.e., providing unrelated explanations), 15 responses are positive to indicate the relatedness of recommended posts. In Q2, the majority (i.e.,19/20 with larger than 3') agrees that SVulD's recommended posts provide the reasons (root cause) for problematic codes. Besides, all responses (i.e., $\geq$ 3) are positive with the *suggestion*. However, about half of the participants give less than 3 scores to the *impacts* of problematic code, which is consistent with our manually labeled data (the impacts of problematic code have the least number). Meanwhile, everyone believes that giving the explanation of *root cause* is most important for explaining a problematic code. In Q3, 13 participants agree that the explanation extracted by SVulD can help them intuitively understand the vulnerable code and the remaining 7 responses have negative feedback, which also confirms the concealment of vulnerability. In Q4, we find that 13 responses rank at top-3 (5 for top-1, 5 for top-2, and 3 for top-3), and 3 responses are scored with 0, which means that none of the recommended explanations are related to the vulnerable function. Finally, in Q5, we use Mean Average Precision (MAP) [27] to qualify the gap between our recommendation and developers' expectations. We get 0.565 of MAP, which means SVulD, to some degree, can give an acceptable recommendation list.

We analyze the negative responses about SVulD and find that the biggest problem falls into the completeness of our dataset, as SVulD cannot find the most semantic similar problematic codes with vulnerable functions in the built dataset (i.e., similarity < 0.5).

Answer to RQ-4: *Our user study reveals, to some extent, that SVulD presents the potential feasibility of assisting developers to intuitively understand the detected vulnerability.*

6 DISCUSSION

This section discusses open questions regarding the performance and threads to the validity of SVulD.

6.1 Why SVulD Outperforms Existing Baselines?

DL-based vulnerability detection approaches have a strong ability to learn a feature representation to distinguish vulnerable functions and non-vulnerable ones. Therefore, the efficacy of the models' vulnerability detection depends largely on how separable the feature representation of the two types of functions (i.e., vulnerable and non-vulnerable) are. The greater the separability of the two functions, the easier it is for a model to distinguish between them.

We adopt Principal Components Analysis (PCA) [6] to inspect the separability of the studied models. PCA is a popular dimensionality reduction technique and is suited for projecting the original feature embedding into two principal dimensional embeddings. Besides, we randomly sample the same number of non-vulnerable functions with vulnerable functions in the testing dataset for more clear visualization.

Fig. 7 illustrates the separability of the studied approaches. From the visualization results (Fig. 7(a)–(d)), we can see that the majority of the functions are mixed and the boundary of each function is not clear, which indicates the difficulty of baselines in drawing the decision boundary. In contrast, UniXcoder (shown in Fig. 7(e)) has better separability than baselines, which indicates the large-scale pre-trained language model (specially trained on C/C++ codes) has a stronger ability to understand the semantic of codes. Lastly, Fig. 7(f) shows the separability of our SVulD. We can observe that SVulD has the best performance in distinguishing vulnerable functions from non-vulnerable ones. Equipped with contrastive learning, SVulD can learn better semantic embedding of functions.

6.2 Threats to Validity

Threats to Internal Validity mainly correspond to the potential mistakes in the implementation of our approach and other baselines. To minimize such a threat, we first implement our model by pair programming and directly utilize the pre-trained models for building vulnerability detectors. We also use the original source code of baselines from the GitHub repositories shared by corresponding authors and use the same hyperparameters in the original papers. The authors also carefully review the experimental scripts to ensure their correctness.

Threats to External Validity mainly correspond to the studied dataset. Even though we have evaluated models on those widely used vulnerability datasets in literature to ensure a fair comparison

with baselines, the diversity of projects is also limited in the following aspects. Firstly, all the studied projects (i.e., functions) are developed in C/C++ programming language. Therefore, projects developed in other popular programming languages (e.g., Java and Python) have not been considered. Secondly, all the studied datasets are collected from open-source projects, and the performance of SVulD on commercial projects is unknown. Thus, more diverse datasets should be collected and explored in future work.

Threats to Construct Validity mainly correspond to the performance metrics used in our evaluations. To minimize such a threat, we adopt a few performance metrics widely used in existing work. In particular, we totally consider five performance metrics including Accuracy, Precision, Recall, F1-score, and PR-AUC.

7 RELATED WORK

7.1 AI-Based Software Vulnerability Detection

Software vulnerability detection has attracted much attention from researchers and many DL-based approaches have been proposed to automatically learn the vulnerability patterns from historical data [8, 16, 28, 30, 31, 44, 45], since the powerful learning ability of deep neural networks has been verified in many software engineering scenarios [32, 33, 46] (e.g., defect prediction, defect repair).

Dam et al. [13] proposed a vulnerability detector with LSTM-based architecture. Russell et al. [38] proposed another RNN-based architecture to automatically extract features from source code for vulnerability detection. However, these approaches assume source code is a sequence of tokens, which ignores the graph structure of the source code. Therefore, Li et al. [29, 30] sequentially proposed two slice-based vulnerability detection approaches, VulDeePecker [30] and SySeVR [29], to learn the syntax and semantic information of vulnerable code. Following that, many graph neural network (GNN) based models [44, 45] are proposed. Cheng et al. [11] proposed DeepWukong by embedding both textual and structural information of code into a comprehensive code representation. Wang et al. [42] proposed FUNDED by combining nine mainstream graphs. Cao et al. [7] proposed MVD to detect fine-grained memory-related vulnerability.

Apart from the coarse-grained models (e.g., function level), researchers also proposed many fine-grained models. Li et al. [28] proposed VulDeeLocator by adopting a program slicing technique to narrow down the scope of vulnerability-prone lines of code. Fu et al. [19] proposed LineVul by leveraging the attention mechanism inside the BERT architecture for line-level vulnerability detection. Hin et al. [23] proposed LineVD to formulate statement-level vulnerability detection as a node classification task.

Different from previous work, our paper focuses on the effective semantic embedding of functions, especially those that are lexically similar.

7.2 Interpretation for AI-Based Software Vulnerability Detection

Developing explainable models is one of the ways for vulnerability detection, which could provide a fine-grained vulnerability prediction outcome. Specifically, many works have attempted to detect line-level information by leveraging explainable AI for software engineering tasks, such as detecting source code lines for defect prediction [33, 37]. This raises the importance of research for interpretable AI-based models.

However, existing studies are limited to providing partial information for the explanation generation. Zou et al. [47] introduced a high-fidelity token-level explanation framework, which aims at identifying a small number of tokens that make significant contributions to a detector's prediction. Li et al. [28] proposed VulDeeLocator to simultaneously achieve high detection capability and high locating precision and it explains detection results at intermediate code. Ding et al. [15] proposed a statement-level model via localizing the specific vulnerable statements with the assumption of receiving vulnerable source codes at the function level. Li et al. [27] adopted explainable GNN to propose IVDetect and provided fine-grained interpretations. Fu et al. [19] proposed a transformer-based line-level model named LineVul and leveraged the attention mechanism of BERT architecture to explain the vulnerable code lines. Recently, Sun et al. [40] conducted the first research work on the application of Explainable AI in silent dependency alert prediction, which opens the door to the related domains.

Different from existing works that focus on explaining why AI-models give out the predicted results, our paper aims at making an explanation for the detected results by providing a develop-oriented natural language described explanation in order to heuristically help developers understand the root cause of the detected vulnerabilities.

8 CONCLUSION AND FUTURE WORK

This paper proposes a novel approach SVulD, which is a function-level subtle semantic embedding for vulnerability detection along with heuristic explanations, technically based on pre-trained semantic embedding as well as contrastive learning. SVulD firstly adopts contrastive learning to train the UniXcoder semantic embedding model for learning distinguishing semantic representation of functions regardless of their lexically similar information. SVulD secondly builds a knowledge-based crowdsource dataset by crawling problematic codes in Stack Overflow to provide developers with heuristic explanations of the detected problematic codes. The experimental results show the effectiveness of SVulD by comparing it with four SOTA deep learning-based approaches.

Our future work will investigate the generalization of contrastive learning to existing deep learning approaches for vulnerability detection.

9 DATA AVAILABILITY

The replication of this paper is publicly available [5].

ACKNOWLEDGEMENTS

This research is supported by the National Natural Science Foundation of China (No. 62202419), the Fundamental Research Funds for the Central Universities (No. 226-2022-00064), the Ningbo Natural Science Foundation (No. 2022J184), and the State Street Zhejiang University Technology Center.

REFERENCES

[1] 2023. Checkmarx. https://www.checkmarx.com/
[2] 2023. FlawFinder. https://dwheeler.com/flawfinder/
[3] 2023. Hugging Face. https://huggingface.co
[4] 2023. Joern. https://github.com/joernio/joern

[5] 2023. Replication. https://github.com/jacknichao/SVulD
[6] Hervé Abdi and Lynne J Williams. 2010. Principal component analysis. *Wiley interdisciplinary reviews: computational statistics* 2, 4 (2010), 433–459.
[7] Sicong Cao, Xiaobing Sun, Lili Bo, Rongxin Wu, Bin Li, and Chuanqi Tao. 2022. MVD: Memory-Related Vulnerability Detection Based on Flow-Sensitive Graph Neural Networks. *arXiv preprint arXiv:2203.02660* (2022).
[8] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2021. Deep learning based vulnerability detection: Are we there yet. *IEEE Transactions on Software Engineering* (2021).
[9] Qibin Chen, Jeremy Lacomis, Edward J Schwartz, Graham Neubig, Bogdan Vasilescu, and Claire Le Goues. 2022. Varclr: Variable semantic representation pre-training via contrastive learning. In *Proceedings of the 44th International Conference on Software Engineering*. 2327–2339.
[10] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. 2020. A simple framework for contrastive learning of visual representations. In *International conference on machine learning*. PMLR, 1597–1607.
[11] Xiao Cheng, Haoyu Wang, Jiayi Hua, Guoai Xu, and Yulei Sui. 2021. Deepwukong: Statically detecting software vulnerabilities using deep graph neural network. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 3 (2021), 1–33.
[12] Xiao Cheng, Guanqin Zhang, Haoyu Wang, and Yulei Sui. 2022. 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*. 519–531.
[13] Hoa Khanh Dam, Truyen Tran, Trang Pham, Shien Wee Ng, John Grundy, and Aditya Ghose. 2017. Automatic feature learning for vulnerability prediction. *arXiv preprint arXiv:1708.02368* (2017).
[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).
[15] Yangruibo Ding, Sahil Suneja, Yunhui Zheng, Jim Laredo, Alessandro Morari, Gail Kaiser, and Baishakhi Ray. 2021. VELVET: a noVel Ensemble Learning approach to automatically locate VulnErable sTatements. *arXiv preprint arXiv:2112.10893* (2021).
[16] Xu Duan, Jingzheng Wu, Shouling Ji, Zhiqing Rui, Tianyue Luo, Mutian Yang, and Yanjun Wu. 2019. VulSniper: Focus Your Attention to Shoot Fine-Grained Vulnerabilities.. In *IJCAI*. 4665–4671.
[17] Gang Fan, Rongxin Wu, Qingkai Shi, Xiao Xiao, Jinguo Zhou, and Charles Zhang. 2019. Smoke: scalable path-sensitive memory leak detection for millions of lines of code. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 72–82.
[18] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. A C/C++ code vulnerability dataset with code changes and CVE summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 508–512.
[19] Michael Fu and Chakkrit Tantithamthavorn. 2022. LineVul: A Transformer-based Line-Level Vulnerability Prediction. (2022).
[20] Tianyu Gao, Xingcheng Yao, and Danqi Chen. 2021. Simcse: Simple contrastive learning of sentence embeddings. *arXiv preprint arXiv:2104.08821* (2021).
[21] The Tcpdump Group. 2023. tcpdump. https://github.com/the-tcpdump-group/tcpdump
[22] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. *arXiv preprint arXiv:2203.03850* (2022).
[23] David Hin, Andrey Kan, Huaming Chen, and M Ali Babar. 2022. LineVD: Statement-level Vulnerability Detection using Graph Neural Networks. *arXiv preprint arXiv:2203.05181* (2022).
[24] Elad Hoffer and Nir Ailon. 2015. Deep metric learning using triplet network. In *International workshop on similarity-based pattern recognition*. Springer, 84–92.
[25] Secure Software Inc. 2023. Rough Auditing Tool for Security (RATS). https://code.google.com/p/rough-auditing-tool-for-security/
[26] Wen Li, Haipeng Cai, Yulei Sui, and David Manz. 2020. PCA: memory leak detection using partial call-path analysis. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1621–1625.
[27] Yi Li, Shaohua Wang, and Tien N Nguyen. 2021. 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*. 292–303.
[28] Zhen Li, Deqing Zou, Shouhuai Xu, Zhaoxuan Chen, Yawei Zhu, and Hai Jin. 2021. Vuldeelocator: a deep learning-based fine-grained vulnerability detector. *IEEE Transactions on Dependable and Secure Computing* (2021).
[29] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. 2021. Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing* (2021).
[30] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. Vuldeepecker: A deep learning-based system for vulnerability detection. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium*.
[31] Guanjun Lin, Jun Zhang, Wei Luo, Lei Pan, and Yang Xiang. 2017. POSTER: Vulnerability discovery with function representation learning from unlabeled projects. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2539–2541.
[32] Martin Monperrus. 2018. Automatic software repair: a bibliography. *ACM Computing Surveys (CSUR)* 51, 1 (2018), 1–24.
[33] Chao Ni, Wei Wang, Kaiwen Yang, Xin Xia, Kui Liu, and David Lo. 2022. The Best of Both Worlds: Integrating Semantic Features with Expert Features for Defect Prediction and Localization. In *Proceedings of the 2022 30th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 672–683.
[34] Chao Ni, Kaiwen Yang, Xin Xia, David Lo, Xiang Chen, and Xiaohu Yang. 2022. Defect Identification, Categorization, and Repair: Better Together. *arXiv preprint arXiv:2204.04856* (2022).
[35] Georgios Nikitopoulos, Konstantina Dritsa, Panos Louridas, and Dimitris Mitropoulos. 2021. CrossVul: a cross-language vulnerability dataset with commit data. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1565–1569.
[36] Aaron van den Oord, Yazhe Li, and Oriol Vinyals. 2018. Representation learning with contrastive predictive coding. *arXiv preprint arXiv:1807.03748* (2018).
[37] Chanathip Pornprasit and Chakkrit Tantithamthavorn. 2022. DeepLineDP: Towards a Deep Learning Approach for Line-Level Defect Prediction. *IEEE Transactions on Software Engineering* (2022).
[38] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. 2018. Automated vulnerability detection in source code using deep representation learning. In *Proceedings of the 17th IEEE international conference on machine learning and applications*. IEEE, 757–762.
[39] Jiamou Sun, Zhenchang Xing, Hao Guo, Deheng Ye, Xiaohong Li, Xiwei Xu, and Liming Zhu. 2022. Generating informative CVE description from ExploitDB posts by extractive summarization. *ACM Transactions on Software Engineering and Methodology (TOSEM)* (2022).
[40] Jiamou Sun, Zhenchang Xing, Qinghua Lu, Xiwei (Sherry) Xu, Liming Zhu, Thong Hoang, and Dehai Zhao. 2023. Silent Vulnerable Dependency Alert Prediction with Vulnerability Key Aspect Explanation. In *IEEE/ACM 45st International Conference on Software Engineering (ICSE)*. IEEE.
[41] Synopsys. 2023. Coverity. https://scan.coverity.com/
[42] Huanting Wang, Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, Dingyi Fang, Yansong Feng, Lizhong Bian, and Zheng Wang. 2020. Combining graph-based learning with automated data collection for code vulnerability detection. *IEEE Transactions on Information Forensics and Security* 16 (2020), 1943–1958.
[43] Yueming Wu, Deqing Zou, Shihan Dou, Wei Yang, Duo Xu, and Hai Jin. 2022. VulCNN: An Image-inspired Scalable Vulnerability Detection System. (2022).
[44] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 590–604.
[45] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *In Proceedings of the 33rd International Conference on Neural Information Processing Systems*. 10197–10207.
[46] Yuxiang Zhu and Minxue Pan. 2019. Automatic code summarization: A systematic literature review. *arXiv preprint arXiv:1909.04352* (2019).
[47] Deqing Zou, Yawei Zhu, Shouhuai Xu, Zhen Li, Hai Jin, and Hengkai Ye. 2021. Interpreting deep learning-based vulnerability detector predictions based on heuristic searching. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 2 (2021), 1–31.

Received 2023-02-02; accepted 2023-07-27