Available online at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/cose

Computers & Security

VDSimilar: Vulnerability detection based on code similarity of vulnerabilities and patches

Hao Sun[a,b], Lei Cui[a,*], Lun Li[a,*], Zhenquan Ding[a], Zhiyu Hao[a], Jiancong Cui[a,c], Peng Liu[d]

[a] Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
[b] School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China
[c] School of Information Science and Engineering, Shandong Normal University, Jinan, China
[d] School of Computer Science and Information Technology, Guangxi Normal University, Guilin, China

ARTICLE INFO

Article history:
Received 18 November 2020
Revised 26 April 2021
Accepted 23 July 2021
Available online 30 July 2021

Keywords:
Siamese network
BiLSTM
Attention
Vulnerability detection
Code similarity

ABSTRACT

Vulnerability detection using machine learning is a hot topic in improving software security. However, existing works formulate detection as a classification problem, which requires a large set of labelled data while capturing semantical and syntactic similarity. In this work, we argue that similarity in the view of vulnerability is the key in detecting vulnerabilities. We prepare a relatively smaller data set composed of both vulnerabilities and associated patches, and attempt to realize security similarity from (i) the similarity between pair of vulnerabilities and (ii) the difference between a pair of vulnerability and patch. To achieve this, we setup the detection model using the Siamese network cooperated with BiLSTM and Attention to deal with source code, Attention network to improve the detection accuracy. On a data set of 876 vulnerabilities and patches of OpenSSL and Linux, the proposed model (VDSimilar) achieves about 97.17% in AUC value of OpenSSL (where the Attention network contributes 1.21% than BiLSTM in Siamese), which is more outstanding than the most advanced methods based on deep learning.

1. Introduction

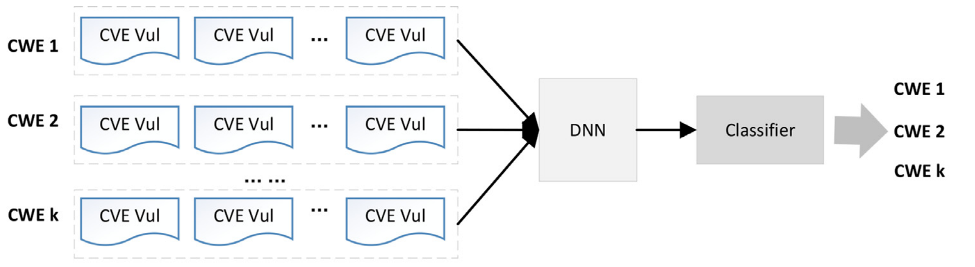
Vulnerability denotes a weakness, defect or security bug in a software program. It is introduced due to design error, low coding quality, or insufficient security testing. The vulnerability can be directly used by a hacker to gain access to a system or network. Thus, vulnerability detection is essential in improving software security.

Code similarity is one promising method to detect vulnerabilities when some known vulnerabilities are given. This is because it is recognized that some vulnerable code share the same class of weakness, e.g., memory buffer overflow, improper input validation, out-of-bounds write. The basic idea of code similarity is to compare the currently known vulnerabilities and code of test programs for finding matches. For example, many approaches first transform the code into an unit[1] of intermediate representation (e.g., tokens, trees, or graphs), and then employ a comparison algorithm to find matched units to known vulnerable unit (Hunt and MacIlroy, 1976; Jiang et al., 2007; Kamiya et al., 2002; Kim et al., 2017; Su et al., 2017; Vinyals et al., 2015). However, due to the large variance of source code in both syntax and semantics, they suffer low accuracy and coverage in detect real-world vulnerabilities. Recently, with the rapid development of deep learning, data-driven vulnerability detection approaches have received much Attention. They view vulnerability detection as

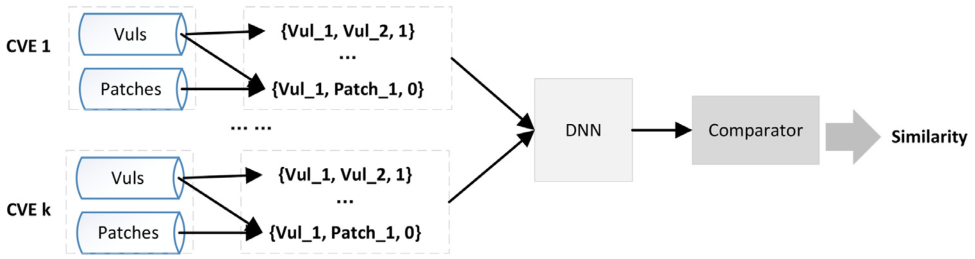
* Corresponding authors.
E-mail addresses: sunnhao@iie.ac.cn (H. Sun), cuilei@iie.ac.cn (L. Cui), lilun@iie.ac.cn (L. Li), dingzhenquan@iie.ac.cn (Z. Ding), haozhiyu@iie.ac.cn (Z. Hao), 201711010104@stu.sdnu.edu.cn (J. Cui), liupeng@gxnu.edu.cn (P. Liu).

https://doi.org/10.1016/j.cose.2021.102417
0167-4048/© 2021 Elsevier Ltd. All rights reserved.

[1] Within the paper, we focus on C/C++ vulnerabilities at the granularity of function.



(a) Traditional DNN approach.



(b) Our approach.

Fig. 1 – Comparison of traditional approach and our approach.

a classification problem and train a classifier on vulnerabilities databases such as CVE Details, NVD, and SARD. In addition, some recent works attempt to employ machine learning methods to compute the similarity between two units transformed from the source code, e.g., graphs (Li et al., 2019). Contributed to the advance of deep learning, these methods have shown the effectiveness in vulnerability detection.

Unfortunately, current code similarity based methods intend to detect vulnerability that is semantically and syntactically similar to a known vulnerability. However, semantical and syntactic similarity does not guarantee to find vulnerabilities. This is because the vulnerable snippet often takes a little fragment of the entire vulnerable function, i.e., the snippet may only contain a few lines or even one line of code, while the function involves dozens to hundreds of lines. Therefore, for two functions that are likely to identical in syntax and semantics, even subtle differences can lead to different categories in the view of vulnerability (i.e., one is vulnerable while another is benign), as we will detail in Fig. 2. Thus, the main property that we want a good detection solution to satisfy is to capture the similarity in the view of vulnerability, rather than simply syntax and semantics.

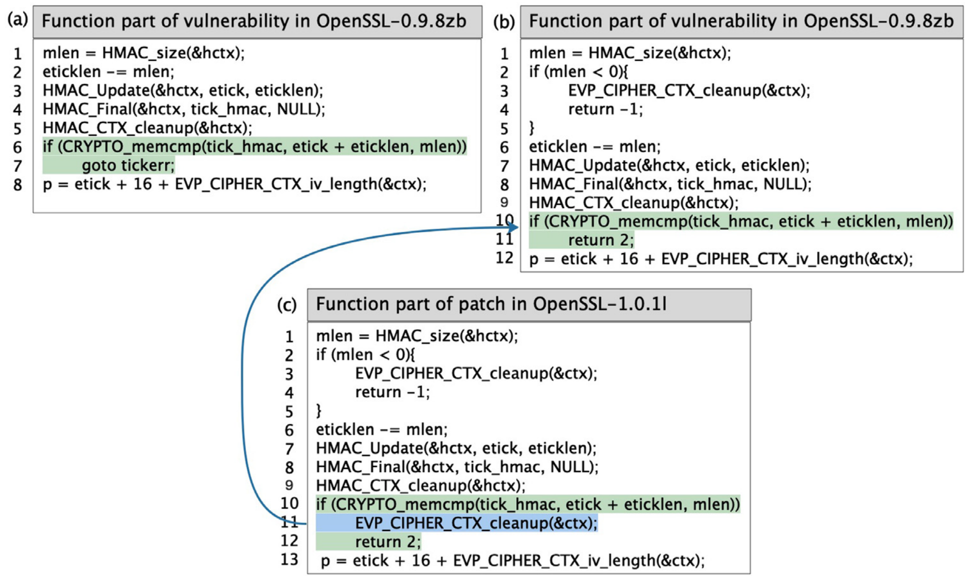
On the other hand, deep learning-based methods always require a large data set to train a model. For example, VulDeePecker Li et al. (2018b) is performed on 61,638 code gadgets for vulnerability detection. Although it is possible to prepare such a large data set, it requires heavy human efforts. Thus, we want a method that can perform deep learning-based detection on a relatively small data set.

1.1. *Our approach*

In this paper, we satisfy the property by presenting a metric learning-based approach, which trains a code similarity detector on a data set of vulnerabilities and patches. In particular, we pay our efforts in two directions, a data set that characterizes vulnerabilities and a metric learning model that learns similarity in the view of vulnerability. First, we prepare a data set of CVE bunches,[2] each of which contains multiple vulnerable functions and patched functions associated with one CVE, as illustrated in Fig. 1. These functions for each CVE can be acquired from various versions of the affected software program, and they follow two rules.

a) For two vulnerable functions of the same CVE across two program versions, the vulnerability snippet will keep despite of the code change. b) For one vulnerable function and the associated patched function, the vulnerability snippet will disappear no matter how the code changes. Therefore, each bunch of functions potentially provide vulnerability characteristics of one CVE. Second, in the view of vulnerability, two vulnerable functions across different versions should be treated as similar, even they experience code changes. On the other hand, a vulnerable function and its patched function,

[2] In general, a reported vulnerability is associated with a CVE (Common Vulnerabilities and Exposures), and each CVE is categorized into one CWE (Common Weakness Emulation) which denotes a class of weakness. We will introduce these terms in detail later.



(a) Function part of vulnerability in OpenSSL−0.9.8zb

```
mlen = HMAC_size(&hctx);
eticklen −= mlen;
HMAC_Update(&hctx, etick, eticklen);
HMAC_Final(&hctx, tick_hmac, NULL);
HMAC_CTX_cleanup(&hctx);
if (CRYPTO_memcmp(tick_hmac, etick + eticklen, mlen))
      goto tickerr;
p = etick + 16 + EVP_CIPHER_CTX_iv_length(&ctx);
```

(b) Function part of vulnerability in OpenSSL−0.9.8zb

```
mlen = HMAC_size(&hctx);
if (mlen < 0){
      EVP_CIPHER_CTX_cleanup(&ctx);
      return −1;
}
eticklen −= mlen;
HMAC_Update(&hctx, etick, eticklen);
HMAC_Final(&hctx, tick_hmac, NULL);
HMAC_CTX_cleanup(&hctx);
if (CRYPTO_memcmp(tick_hmac, etick + eticklen, mlen))
      return 2;
p = etick + 16 + EVP_CIPHER_CTX_iv_length(&ctx);
```

(c) Function part of patch in OpenSSL−1.0.1l

```
mlen = HMAC_size(&hctx);
if (mlen < 0){
      EVP_CIPHER_CTX_cleanup(&ctx);
      return −1;
}
eticklen −= mlen;
HMAC_Update(&hctx, etick, eticklen);
HMAC_Final(&hctx, tick_hmac, NULL);
HMAC_CTX_cleanup(&hctx);
if (CRYPTO_memcmp(tick_hmac, etick + eticklen, mlen))
      EVP_CIPHER_CTX_cleanup(&ctx);
      return 2;
 p = etick + 16 + EVP_CIPHER_CTX_iv_length(&ctx);
```

Fig. 2 – Function *tls_decrypt_ticket* across three different OpenSSL versions ((a) Vulnerable in 0.9.8zb.(b) Vulnerable in 1.0.1i.(c) Patched in 1.0.1l.).

even looks similar in syntax, should be treated as different since the vulnerability snippet has been removed. Inspired by this, we employ the Siamese model to learn the similarity between two vulnerable functions, while learning the difference between vulnerable function and patched function. In addition, we incorporate BiLSTM and Attention network in the Siamese model, which helps generate more accurate and focused representations of functions for detection. In this way, the trained model can perceive similarity in the view of vulnerability instead of merely semantics and syntax.

1.2. *Main contributions*

Fig. 1 compares our approach with existing works. First, existing works requires a large set of functions (e.g., about 1M functions in Russell et al., 2018), each of which is labelled as vulnerable or not. In comparison, our prepared data set contains a relatively smaller set of both vulnerable functions and patched functions without labelling. By pairing the functions, we naturally augment the training data. Second, most of existing methods view the vulnerability detection as a classification problem, while we pay attention to the code similarity with a metric learning model incorporated with Attention network. The main advantage of our model is that it can pay attention to the snippets similarity in terms of vulnerability similarity rather than syntactic and semantical similarity of the entire functions.

To summarize, we make the following contributions.

- First, a data set containing real-world vulnerable functions and associated patched functions of 147 OpenSSL and Linux CVEs, which helps characterize vulnerability snippets. The data set is available in Github.[3]
- Second, a detection model using Siamese network cooperated with BiLSTM and Attention network. By taking pairs of vulnerable functions and patched functions, the model is able compute the similarity between two functions, so as to detect vulnerabilities. We plan to release the source code upon the publication of this paper.
- Third, the system implementation and evaluation on the data set to prove the effectiveness of the proposed model.

1.3. *Organization of this work*

In the next section, we provide a brief introduction to several important notations and summarize related work from two aspects: traditional methods and machine learning methods. In Section 3, we present the basic idea that uses the vulnerability and patches for the similarity comparison. In Section 4, we describe the technical details of VDSimilar, including data preparation, the detection model of VDSimilar and the process of detecting vulnerabilities. In Section 5, we compare VDSimilar against several existing works to show its effectiveness. In Section 6, we conclude our work and discuss the future work.

2. Background and related work

We first present several important notations used in the paper. Then we give a brief introduction to code similarity-based detection approach and review existing works in this literature.

[3] Data set, https://github.com/sunhao123456789/siamese_dataset.

2.1. Notations

Vulnerability A vulnerability denotes a weakness, defect or security bug in a program which can be exploited by an attacker to perform unauthorized actions within a system.

Vulnerable function A vulnerability may involve one or many functions. E.g., the well-known vulnerability HeartBleed involves two functions in the OpenSSL program, i.e., *dtls1_process_heartbeat* and *tls1_process_heartbeat*. For brevity we assume that one vulnerability involves one vulnerable function within the paper.

Vulnerability snippet Vulnerability snippet here denotes the real code lines that compose the vulnerability. Generally, the snippet takes only a small part of the entire vulnerability, as we will show in Fig. 2.

Patch A patch denotes pieces of code that repair the vulnerable function, and it involves addition, removal and modification of code. Generally, each patched function is associated with one vulnerable function.

CVE Common Vulnerabilities and Exposures (CVE) is a catalog of known security threats. In general, a CVE denotes a reported vulnerability within a program, e.g., CVE-2014-0160 (the HeartBleed vulnerability) refers to *overflow obtain information* threat in *OpenSSL 1.0.1*.

CWE Common Weakness Emulation (CWE), is a class of weakness that can lead to a vulnerability. It denotes the underlying law of a set of vulnerabilities. E.g., CVE-2014-0160 is categorized into CWE-119, which denotes *Improper Restriction of Operations within the Bounds of a Memory Buffer*.

2.2. Code similarity based detection

Traditional approaches Many existing studies abstract the program code into a high level representation so that various code are identical at the high level, and then employ comparison algorithms to find matched code. For example, Sajnan et al. divide the code into smaller code blocks and represent each block with a bag of tokens. Then, they find similar blocks by querying from a partial inverted index (Kamiya et al., 2002). Jiang et al. represent the code with abstract syntax tree (AST) for each file and extract characteristic vectors from the ASTs. Then, they compute the similarity between vectors by Euclidean distance to find similar code (Jiang et al., 2007). Li et al. characterize source code with control flow graph (CFG) and employ isomorphism matching on graphs to find identical code. They also propose four optimizations to improve the comparison performance (Li et al., 2019). Roy et al. propose NICAD to parse code text with potential similar code extractor and standard pretty-printer and compare potential codes by clustering (Roy and Cordy, 2008). Jang et al. propose ReDeBug to slide a window of N tokens and then apply K independent hash functions to each window (Jang et al., 2012). Finally, ReDeBug uses the Bloom Filter method to find matches.

Although these methods show effectiveness in finding code with similar patterns, the abstraction of code would lead to loss of semantics and vulnerability features, which makes it difficult to detect vulnerabilities with high accuracy.

Machine learning-based approaches In recent years, as machine learning methods are being successfully applied in many scenarios, several studies employ machine learning to identify vulnerabilities. For example, Li et al. extract a set of manually selected features from vulnerabilities and patches, train several classifiers using different learning methods, and automatically choose the most suitable one in detecting vulnerabilities (Li et al., 2016). Grieco et al. propose a method that automatically extracts a set of both static features and dynamic features from a large code set and then employs random forest model to train a classifier (Grieco et al., 2016). Xu et al. use CFG to represent the function of program binaries and extract a set of features related to code instructions and graph structures, they then employ the Siamese network to compare two CFGs generated from the same function with different compilation configurations (Xu et al., 2017). Li et al. considers program representation from three perspectives: Syntax, Semantics and Vector Representations(SySeVR). Based on the existing large number of data sets, NVD and SARD (2021) use BGRU (Graves and Schmidhuber, 2005; Hochreiter and Schmidhuber, 1997) for training to detect whether the specified file has vulnerabilities (Li et al., 2018a). Li et al. generate code gadgets from program dependency graphs and transform each code gadget into a vector of symbolic representation. They train a BiLSTM neural network on vectors for vulnerability classification (Li et al., 2018b). Li et al. propose a graph matching network model to train directly on CFGs for detecting vulnerabilities in program binaries (Li et al., 2019). Zou et al. improves (Li et al., 2018b) with code attention to pinpoint the types of vulnerabilities, so that the deep learning model supports multiclass vulnerability detection (Zou et al., 2019). Li et al. propose VulDeeLocator that is trained on intermediate code rather than native source code to accommodate semantic information (Li et al., 2020). Meanwhile, VulDeeLocator can pin fine-grained vulnerabilities with the use of attention. Liang et al. extract CFG for binary functions and utilize neural network model to locate the potential vulnerable functions, then they leverage bipartite graph matching upon three-level features between two binary functions for vulnerability detection (Liang et al., 2020). Mazuera–Rozo et al. show that most methods perform well on synthetic examples of vulnerabilities yet struggle to achieve very high performance on real-world data set (Mazuera-Rozo et al., 2021). Chakraborty et al. also show that many existing methods perform poor in real-world vulnerabilities and list several suggestions that help to improve the performance (Chakraborty et al., 2020). Feng et al. employ AST to acquire all syntax features and produce a vector to characterize these features (Feng et al., 2020). In addition, they leverage the pack-padded method on the Bi-GRU network which takes as input variable-length data directly so as to learn full features. Wang et al. transform source code into graphs and propose a graph-based learning model on top of graph neural network, which can capture the program syntax, semantics and flows upon training (Wang et al., 2020). He et al. propose to use few-shot learning model to discover code clones, which facilitates to detect vulnerabilities (He et al., 2020).

These methods improve the accuracy and coverage in detecting similar patterns. However, two functions with similar syntax or semantics don't necessarily guarantee to find vulnerability, since the vulnerability snippet only takes a little part of the entire vulnerable functions. Therefore, how to

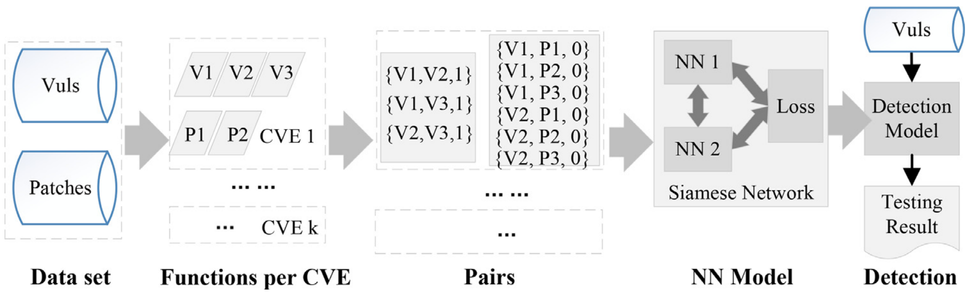


Fig. 3 – Overview architecture.

detect similar code in the view of vulnerability (i.e., detect vulnerability snippets) effectively is still a challenging problem for vulnerability detection. In addition, methods based on deep learning rely on a large data set for training, and cannot accurately detect vulnerabilities when the amount of data is limited.

3. Basic idea

3.1. *Why similarity of vulnerability snippet*

We first explain why the snippet rather than the entire function is the key to detect vulnerabilities.

Fig. 2 shows a code fragment of *tls_decrypt_ticket* function of *t1_lib.c* evolved across three versions of OpenSSL. This function is reported as a vulnerability (CVE-2014-3567) that affects 0.9.8zb and 1.0.1i, and then is fixed in the later version 1.0.1l. As can be seen, compared to Fig. 2(a), the fragment in Fig. 2(b) adds 4 lines of code (lines 2–5) and changes 1 line (line 11), a total of 5 lines of code differences. In contrast, Fig. 2(b) and (c) differ by only one line of code, i.e., line 11 in Fig. 2(c). Thus, Fig. 2(b) is more similar to Fig. 2(c) in terms of both syntax and semantic than Fig. 2(a). However, this conclusion is contrary to the identification of vulnerability, as Fig. 2(a) and (b) show the same vulnerability. The reason is as follows.

The OpenSSL program evolves continuously (e.g., 13 versions are released in 2018), where a function may be modified for many purposes such as bug fixing, performance optimization, or code reconstruction. Therefore, the same function across two versions may show large difference in syntax and semantics. On the other hand, for a vulnerability needed to be fixed, the patches may involve only a few or even one code line, so that the vulnerable functions and patched functions across consecutive versions are highly similar in syntax.

Thus, in detecting vulnerabilities, a good code similarity-based method should pay more attention to the similarity of vulnerability snippets, i.e., code fragments in the view of vulnerability similarity, rather than syntactic similarity and semantical similarity of the entire functions.

3.2. *Design*

To detect similar code in the view of vulnerability, we pay our efforts in two directions, as shown in Fig. 3.

First, a data set that helps characterize vulnerability snippets The differences between vulnerable function and patched function potentially pinpoint the fragment where the vulnerability snippet is potentially located. For example, the bright part in Fig. 2(c) has more blue parts than the bright part in Fig. 2(b), so as to avoid some sensitive information being left in memory and causing security hidden offence. However, the fragment may also contain vulnerability-agnostic code which are used to improve code quality or add functionality. Therefore, more fragments are required to effectively characterize vulnerabilities.

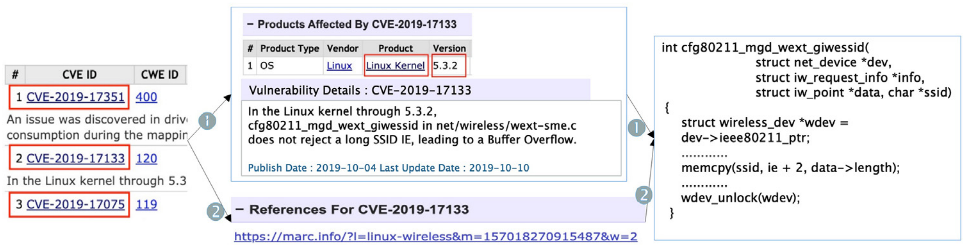
To this end, for each CVE, we collect vulnerable functions across different program versions that are affected by the vulnerability (e.g., $V1$, $V2$ and $V3$ for $CVE1$ shown in Fig. 3), as well as patched functions from program versions where the weakness has been fixed (e.g., $P1$ and $P2$ for $CVE1$). Finally, we prepare a data set for providing vulnerability characterizes.

Second, a metric learning model to train a detector Given vulnerable functions and patched functions, a natural approach is to train a classifier to distinguish between the two. Unfortunately, the number of functions for each CVE is only a few, which is insufficient for training a good classifier. E.g., 94.36% of Linux CVEs have no more than 10 diverse vulnerable functions across all versions. Therefore, we propose to employ the metric learning model (e.g., Siamese network within the paper) to exploit the code similarity.

As can be seen in Fig. 3, for a CVE, the Learning model learns to compute similarity from both similarity pairs and difference pairs. The similarity pair labelled as 1 is a pair of two vulnerable functions, e.g., $\{V1, V2, 1\}$, while the difference pair labelled as 0 is composed of a vulnerable function and a patched function, e.g., $\{V1, P2, 0\}$). Consider that vulnerability snippets only takes a little part of the entire function, we incorporate the Attention network to the Siamese network to focus on the really vulnerable snippets. Finally, with a set of vulnerable functions and patched functions of multiple CVEs, the detection model will learn whether two codes are similar or not in the view of vulnerability. Upon test, the model will compute similarity between a test function and known vulnerable functions, and report the test function for further investigation if it is highly similar to one vulnerable function.

4. Implementation details of VDSimilar

In this section, we will describe several technical details, including data preparation, detection model setup, and vulnerability detection.



#	CVE ID	CWE ID
1	CVE-2019-17351	400
An issue was discovered in driv consumption during the mappin		
2	CVE-2019-17133	120
In the Linux kernel through 5.3		
3	CVE-2019-17075	119

– Products Affected By CVE-2019-17133

#	Product Type	Vendor	Product	Version
1	OS	Linux	Linux Kernel	5.3.2

Vulnerability Details : CVE–2019–17133

In the Linux kernel through 5.3.2, cfg80211_mgd_wext_giwessid in net/wireless/wext-sme.c does not reject a long SSID IE, leading to a Buffer Overflow.

Publish Date : 2019–10–04 Last Update Date : 2019–10–10

– References For CVE-2019-17133

https://marc.info/?l=linux-wireless&m=157018270915487&w=2

```
int cfg80211_mgd_wext_giwessid(
                    struct net_device *dev,
                    struct iw_request_info *info,
                    struct iw_point *data, char *ssid)
{
    struct wireless_dev *wdev =
    dev->ieee80211_ptr;
    ............
    memcpy(ssid, ie + 2, data->length);
    ............
    wdev_unlock(wdev);
}
```

Fig. 4 – The data preparation processing. (1) Locate the vulnerability function according to the function name, file name and program of affected versions. (2) Extract the vulnerability function from patches in external link.

4.1. *Data preparation*

To prepare the data set, we collect a set of CVEs from the CVE Details database.[4] Generally, the vulnerabilities can be acquired in two ways, as shown in Fig. 4. 1) Download the affected products by CVE-2019-17133, and then extract the function of *cfg80211_mgd_wext_giwessid* following the file name and the function name described in vulnerability details. 2) Extract the vulnerabilities directly from patches that are referred in the external link. Since the external links may be unavailable, here we use the first method to extract vulnerabilities.

In order to obtain patches, we assume that the versions of Linux and OpenSSL after the latest affected version have been repaired, e.g., the *cfg80211_mgd_wext_giwessid* functions in versions after Linux-5.3.2 are viewed as patched functions. Finally, by extracting the involved vulnerabilities and patched functions from several versions of programs, we generate a set of similarity pairs $(\{V, V, 1\})$ and difference pairs $(\{V, P, 0\})$. The more detailed steps are described as follows:

Collection of vulnerabilities and programs The CVE Details provide detailed information of the reported vulnerabilities, including associated software programs and affected program versions, involved function name, file name and patches (usually in external links). We employ a web-crawling framework Scrapy to crawl the CVE Details website and extract these information from the description pages of each CVE. Meanwhile, we download all program versions of the involved software. Then, the patched versions can be obtained from all versions released after the newest affected vulnerable version. Finally, we can get a tuple for each CVE, i.e., (CVE, software, affected versions, patched versions, file names, function names).

Extracting vulnerability and patched functions For one CVE, given the detailed information above, it is easy to extract a set of vulnerable functions and patched functions by parsing the source code using LLVM. However, there exist two issues here. First, a function may remain unchanged across consecutive versions, such duplication of functions can artificially affect performance metrics and conceal fitting of training. Second, the descriptions of some CVEs are not accurate and can cause misinformation. E.g., the description of CVE-2015-1792 claims that OpenSSL versions of 1.0.2 series before 1.0.2b are affected. However, according to our analysis, versions from 1.0.2b to

[4] Within the paper, we focus on the CVEs of Linux and OpenSSL published in https://www.cvedetails.com/

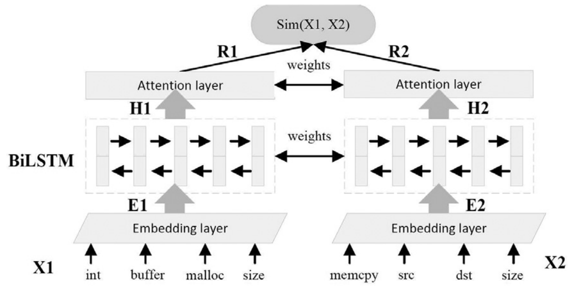


Fig. 5 – The architecture of detection model.

1.0.2o are also affected but not yet reported. This issue will introduce obfuscation upon training since two identical functions are mistakenly labelled as different.

To mitigate these problems, we first compute the hash value of the functions extracted from all program versions. Then, we de-duplicate the functions having the same hash value, thereby solving the first issue. Finally, if the hash value of a claimed benign function (i.e., not listed in the affected versions) is identical to that of a vulnerable function of the same CVE, then it will be labelled as vulnerable, thereby solving the second issue.

Preparing the pairs For each CVE, we prepare two types of pairs, (i) similarity pair which is composed by the combination of any two vulnerable functions from different versions and (ii) difference pair which contains one vulnerable function and one patched function. In this way, although the number of vulnerable functions and patched functions of one CVE is only a few, the number of pairs composed by these functions is relatively large, i.e., $m*(m-1)/2$ similarity pairs and $m*n$ difference pairs for m vulnerable functions and n patched functions, which is sufficient for training a good detection model.

4.2. *Detection model*

We setup the detection model adopting the Siamese architecture which contains two identical sub-networks sharing the same weights (Neculoiu et al., 2016). Each sub-network is composed of Embedding layer, BiLSTM layer and Attention layer, as illustrated in Fig. 5. And the notations of detection model are in Table 1.

The detection model takes two functions as input, each of which is represented by a sequence of words, e.g., $X =$

Table 1 – The notations of detection model.

X	Words vector of functions
S_{X_1,X_2}	Similarity between two functions X_1, X_2
W	Embedding matrix of all words
E	Words embedding matrix of functions
H	Associated words annotations
Q	Queries in the Attention layer
K	Keys in the Attention layer
V	Values in the Attention layer
R	The final representation of functions
D	Distence between two functions
N	Number of function pairs
L	Contrastive loss of two functions

$(x_1, x_2, \ldots, x_m)$ where m denotes the number of words in function.

Embedding layer The sequence X is firstly transformed into a $m*s$ matrix E, where s refers to the embedding size of words and each row denotes the initial embedding of a word. In the experiment, we will discuss the impact of changes in the embedding layer on the code similarity results.

BiLSTM layer Note that there exist correlation between words within a line or across lines of code in a function. Therefore, the representation of the function code should consider the words associated with a word both in forward and backward. Inspired by this, we employ the BiLSTM (Bidirectional LSTM) (Graves et al., 2013) model to deal with the function code. The BiLSTM layer takes as input a matrix E and produces associated word annotations $H = (h_1, h_2, \ldots, h_{2m})$ where h_i denotes the hidden state.

Attention layer As mentioned before, the real vulnerability snippets, which should be paid more attention to, only takes a little fraction of the entire function. First we get the words annotations H from BiLSTM layer. Then initialize three matrix W_Q, W_K, W_V to transform the H to Q, K, V. Then we gain the final representation of the inputs R according to the (1):

$$R_1 = \mathrm{softmax}(Q_1 \cdot K_1^T / \sqrt{d_k}) \cdot V_1 \tag{1}$$

When the word embedding matrix passes through the bidirectional LSTM layer, it can well represent its local features, while the Attention layer allocates attention to local features from the global perspective, so as to arm our model with the ability to focus on its feature subset (Vaswani et al., 2017).

Siamese model Siamese network is a twin networks with shared weights (Chopra et al., 2005) used for metric learning. Here we employ it to learn the similarity between functions. It takes as input similarity pairs (label y is 1) and difference pairs (y is 0), and the goal of training is to maximize the similarity between vulnerable functions meanwhile minimizing the similarity between vulnerable functions and patched functions in the embedding space. Specifically, let S_{X_1,X_2} denote the similarity of a pair $\{X_1, X_2, y\}$, it is calculated by (2)

$$S_{X_1,X_2} = 1 - D \tag{2}$$

where D denotes the Euclidean distance of representations of X_1 and X_2, is calculated by the (3).

$$D = \frac{R_1 \cdot R_2}{\|R_1\| \cdot \|R_2\|} \tag{3}$$

Then, the loss L is calculated by (4).

$$L \leftarrow \frac{1}{2N} \sum_{n=1}^{N} yD^2 + (1 - y)max(margin - D, 0)^2 \tag{4}$$

For a set of one CVE $X = X_i(i = (1, \ldots, N))$, the total loss $L(X)$ is calculated by (5)

$$L(X) = \sum (L(X_i, X_j, y)) \tag{5}$$

where $i, j \in (1 \ldots, N)$ and $i \neq j$. Correspondingly, for the whole data set containing multiple CVEs, the total loss value is the sum of the losses of each CVE set.

4.3. *Detecting vulnerabilities*

Once we have trained a detection model on a set of similar pairs and different pairs, we can use it to compute similarity between testing functions and known vulnerable functions. If the similarity is higher than a predefined threshold, then the testing function will be suspected as potentially vulnerable and reported for further investigation.

Algorithm 1 describes how we compare two functions for

```
Input: Two functions which are represented by sequence
       of words $\mathbf{X_1}, \mathbf{X_2}$
Output: Similarity between two functions $S_{X_1,X_2}$
1  Initialize the embedding matrix of words $W$
2  $E_1 \leftarrow \varnothing, E_2 \leftarrow \varnothing$
3  for $x_1 \in X_1, x_2 \in X_2$ do
4      $e_1 \leftarrow W[x_1], e_2 \leftarrow W[x_2]$
5      append $e_1$ to $E_1$, $e_2$ to $E_2$
6  $H_1 \leftarrow \mathrm{BiLSTM}(E_1), H_2 \leftarrow \mathrm{BiLSTM}(E_2)$
7  $Q_1 \leftarrow H_1 W_{Q_1}, Q_2 \leftarrow H_2 W_{Q_2}$
8  $K_1 \leftarrow H_1 W_{K_1}, K_2 \leftarrow H_2 W_{K_2}$
9  $V_1 \leftarrow H_1 W_{V_1}, V_2 \leftarrow H_2 W_{V_2}$
10 $R_1 = \mathrm{softmax}(Q_1 K_1^T / \sqrt{d_k}) V_1$
11 $R_2 = \mathrm{softmax}(Q_2 K_2^T / \sqrt{d_k}) V_2$
12 $D \leftarrow \frac{R_1 \cdot R_2}{\|R_1\| \times \|R_2\|}$
13 $S_{X_1,X_2} \leftarrow 1 - D$
14 return $S_{X_1,X_2}$
```

Algorithm 1: Detection similarity of two samples.

vulnerability detection. In lines 1 to 5, we first initialize the embedding matrix of the whole corpus W and the embedding matrix of functions E_1, E_2. Then, according to the inputs $X_1$and X_2, which can be served as the indexes of the W, the matrix $E_1$and E_2 are determined. In line 6, the BiLTSM model takes embedding matrix $E_1$and E_2 as inputs, and produces semantic information related to the front and back. From line 7 to 11, the query matrix Q, key matrix K and value matrix V are transformed from the linear mapping of matrix H. Then the final representation of function R is computed according to the

Table 2 – Dataset of OpenSSL and Linux.

	Linux	OpenSSL	Label
CVEs	56	10	
vulnerable functions	369	86	Positive(1)
patched functions	306	115	Negtive(0)
similar pairs	1124	334	Positive(1)
different pairs	2026	672	Negtive(0)

Table 3 – Distribution of vulnerabilities.

CWE	Percent	Description
772	0.013	Missing Release of Resource after Effective Lifetime
476	0.025	NULL Pointer Dereference
416	0.013	Use After Free
399	0.177	Resource Management Errors
362	0.063	Race Condition
310	0.013	Cryptographic Issues
264	0.114	Permissions, Privileges, and Access Controls
200	0.165	Exposure of Sensitive Information to an Unauthorized Actor
190	0.025	Integer Overflow or Wraparound
189	0.127	Numeric Errors
119	0.114	Improper Restriction of Operations within the Bounds of a Memory Buffer
94	0.013	Code Injection
20	0.127	Improper Input Validation
19	0.013	Data Processing Errors

scaled dot-product attention. Finally, the distance between R_1 and R_2 is calculated using Cosine similarity, and the value is predicted as output.

5. Evaluation

5.1. *Experiments setup*

5.1.1. *Data set*

We evaluate the proposed idea on a set of OpenSSL and Linux vulnerabilities. Since several CVEs affect only a few versions, leading to a few pairs whose account is insufficient for training an accurate model. Therefore, we only consider the CVEs that have more than 5 vulnerable functions and 5 patched functions, so that the number of generated similar pairs and different pairs is sufficient for training. This leaves us with 56 Linux CVEs and 10 OpenSSL CVEs, which fall into 14 CWEs, as shown in Table 3. It can be seen that these CVEs contain many types of weakness, e.g., CWE-399 denotes resource management errors, CWE-200 denotes exposure of sensitive information to an unauthorized actor, while CWE-20 denotes improper input validation.

As listed in Table 2, for OpenSSL we obtain a set of 10 CVEs which has 86 vulnerable functions (labelled as positive) and 115 patched functions (labelled as negative). By pair generation, we generate 1612 pairs, of which 334 are similar pairs (labelled as positive) and 1278 are different pairs (labelled as negative), we randomly choose 672 different pairs from 1278 to avoid data imbalance. As for Linux, we obtain a set of 56 CVEs which has 369 vulnerable functions and 306 patched functions. By pair generation, we generate 3150 pairs, of which 1124 are similar pairs and 2026 are different pairs.

5.1.2. *VDSimilar model*

We implement VDSimilar on top of Tensorflow. The default settings of VDSimilar are as follows. The main parameters for BiLSTM network are: embedding size is 64, dropout is 0.2, batch size is 64, number of epochs is 150, dimension of hidden units is 32, number of hidden layers is 2. It uses minibatch stochastic gradient descent with ADAMAX and set learning rate to 0.001. The parameters of Attention layer are: the number of attention head is 2 for Linux date set and 4 for OpenSSL date set, and the dimension of matrix W_Q, W_K, W_V is 32. In addition, we evaluate VDSimilar configured with various settings and compare the performance in Section 5.4.

5.1.3. *Existing tools*

In this paper, we compare VDSimilar with four traditional methods (i.e., Jang et al., 2012; PMD-CPD, 2021; Roy and Cordy, 2008; Simian, 2021 and two deep learning based methods (i.e., SyseVR Li et al., 2018a and VulDeePecker Li et al., 2018b). In addition, we also compare the two variants of VDSimilar, i.e., Siamese-RNN and Siamese-BiLSTM (2021). Both methods use Siamese, but they are configured with simple neural network models without Attention network.

These methods are described as follows:

- Simian (2021). It is a tool for searching duplicated and similar codes.
- Nicad (Roy and Cordy, 2008). The NiCad clone detector is a scalable, flexible clone detection tool designed to implement near-miss intentional clone detection method.
- ReDebug (Jang et al., 2012). It is a system for quickly finding unpatched code clones in OS-distribution scale code bases. It uses a quick, syntax-based approach that scales to OS distribution-sized code bases that include code written in many different languages.
- PMD-CPD (2021). It is an open source static source code analyzer that reports issues in application code. PMD-CPD includes built-in rule sets and supports the ability to write custom rules.
- SyseVR (Li et al., 2018a). It is the first systematic framework for using deep learning to detect vulnerabilities. SySeVR consists of Syntax, Semantics and Vector Representations. It uses BGRU (Graves and Schmidhuber, 2005; Hochreiter and Schmidhuber, 1997) to classify and predict vulnerabilities. It's worth noting that the native SyseVR transforms the source code into graphs, yet here we feed the source code directly.
- VulDeePecker (Li et al., 2018b). It uses code gadgets to represent programs and transform them into vectors, where a code gadget is a number of lines of code that are semantically related to each other (not necessarily consecutive). Similar to SyseVR (Li et al., 2018a), it uses a trained deep learning classification model to make predictions. Here the VulDeePecker model takes as input the source code, rather than the transformed code gadgets.

Table 4 – The notations of metrics.

Metric	Formula	Meaning
FPR	$\frac{FP}{FP+TN}$	The proportion of FP samples in the total samples that are not vulnerable.
FNR	$\frac{FN}{TP+FN}$	The proportion of FN samples in the total samples that are vulnerable.
A	$\frac{TP+TN}{TP+FP+TN+FN}$	The correctness of all detected samples.
P	$\frac{TP}{TP+FP}$	The correctness of detected vulnerable samples.
F1-score	$\frac{2 \cdot P \cdot (1-FNR)}{P+(1-FNR)}$	The overall effectiveness considering both precision and recall.

Table 5 – TP, FN, TN, and FP for OpenSSL data set.

Method	TP	FN	TN	FP
Simian	32	13	48	390
Nicad	40	5	171	267
ReDebug	37	8	97	341
PMD-CPD	31	14	44	394
VulDeePecker	32	13	289	149
SyseVR	41	4	387	51
Siamese-RNN	45	0	426	12
Siamese-Bilstm	45	0	430	8
VDSimilar	45	0	431	7

- Siamese-RNN and Siamese-BiLSTM (2021). They are two variants of VDSimilar, i.e., using Siamese network configured with RNN and BiLSTM respectively.

We use 10-fold cross-validation upon training and testing. Specifically, we randomly divide the data set into 10 subsets. Then, we prepare 10 different groups of data in which 9 subsets are used for training and 1 for testing. Finally, we get 10 groups of results and report the average result. All experiments are performed on a server configured with six Intel Core i7-8700 CPU@3.20 GHz, 16 G of RAM, and 8 GB memory of NVIDIA GeForce RTX 2070.

5.2. *Measurements*

We measure the effectiveness of VDSimilar in terms of widely-used metrics (Pendleton et al., 2016): false positive rate (FPR), false negative rate (FNR), accuracy (A), precision (P), and F1-measure (F1). Let TP (true positive), FP (false positive), TN (true negative), FN (false negative) denote the number of correctly predicted vulnerabilities, falsely predicted vulnerabilities, correctly predicted benign functions, falsely predicted benign functions. The definition of these metrics are shown in Table 4. It's worth noting that smaller FPR and FNR, higher A, P and F1 imply that the model is better.

5.2.1. *Overall results*

We compare VDSimilar with 8 existing methods. We first report the overall results, Tables 5 and 6 report the value of TP, FN, TN and FP of various methods for OpenSSL and Linux respectively, and Tables 7 and 8 show the results of various metrics. From these results, we get the following conclusions.

Table 6 – TP, FN, TN, and FP for Linux data set.

Method	TP	FN	TN	FP
Simian	757	375	341	2490
Nicad	775	357	1244	1587
ReDebug	760	372	569	2262
PMD-CPD	681	451	254	2577
VulDeePecker	743	389	1956	875
SyseVR	782	350	2565	266
Siamese-RNN	1114	18	2754	77
Siamese-Bilstm	1115	17	2771	60
VDSimilar	1120	12	2774	57

Table 7 – Comparison of various methods for OpenSSL data set.

Method	FPR(%)	FNR(%)	A(%)	P(%)	F1(%)
Simian	89.11	29.07	61.65	23.01	20.4
Nicad	60.96	13.33	64.48	27.75	29.1
ReDebug	78.00	19.11	63.58	25.40	26.2
PMD-CPD	90.16	31.67	60.72	21.89	13.1
VulDeePecker	34.22	30.27	36.16	53.33	27.9
SyseVR	11.72	9.68	71.91	85.19	80.7
Siamese-RNN	2.74	1.67	74.21	86.42	85.3
Siamese-Bilstm	2.05	1.45	86.33	90.33	89.9
VDSimilar	1.70	1.37	90.81	91.81	91.4

Table 8 – Comparison of various methods for Linux data set.

Method	FPR(%)	FNR(%)	A(%)	P(%)	F1(%)
Simian	87.99	33.19	61.20	22.01	19.5
Nicad	56.07	31.61	63.68	26.29	28.2
ReDebug	79.92	32.95	62.22	24.31	25.3
PMD-CPD	91.04	39.88	60.56	20.52	12.9
VulDeePecker	30.92	34.39	62.50	40.62	29.2
SyseVR	9.41	30.92	51.92	81.15	80.0
Siamese-RNN	2.72	1.63	70.16	85.16	84.9
Siamese-Bilstm	2.12	1.51	88.07	89.07	88.7
VDSimilar	2.02	1.08	91.75	92.51	91.1

- Traditional methods, i.e., Simian, Nicad, ReDebug and PMD-CPD, suffer high FPR and FNR and achieve low A, P and F1. Deep learning methods perform better than traditional methods, showing that they can better represent the codes and learn the similarity between codes.
- Variants of VDSimilar, i.e., Siamese-RNN and Siamese-Bilstm, both of which employ Siamese network, outperforms VulDeePecker and SyseVR. This is because the data set is relatively small, i.e., only hundreds of vulnerable functions and patched functions. Using such a small data set, existing deep learning based method fail to learn representative features in the view of vulnerabilities. In contrast, the Siamese network generate similar pairs and dif-

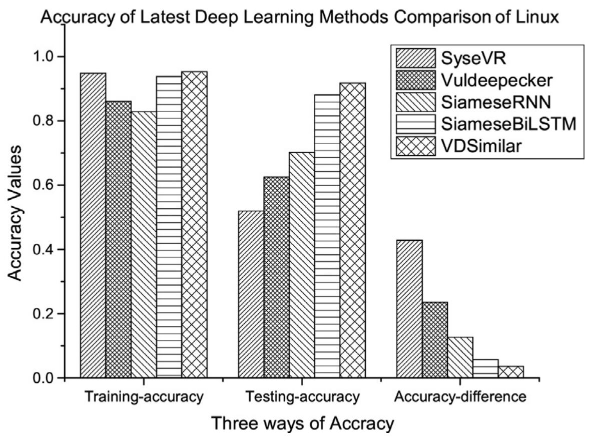


Fig. 6 – Comparison of deep learning models for Linux.

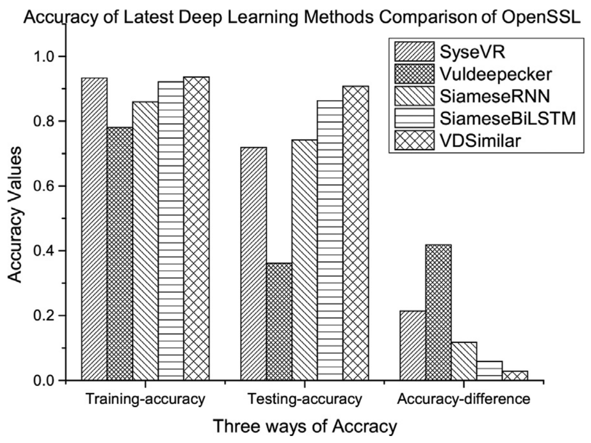


Fig. 7 – Comparison of deep learning models for OpenSSL.

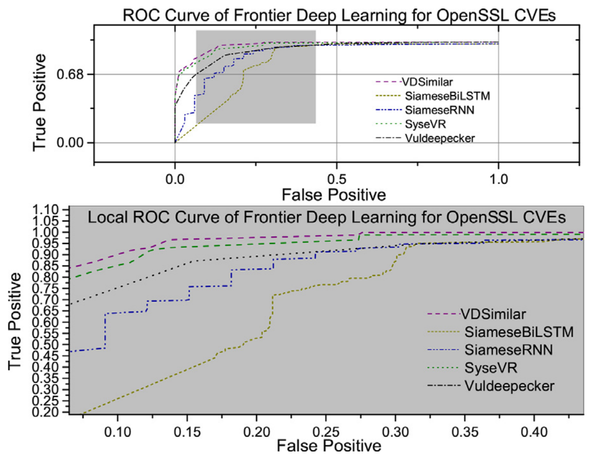


Fig. 8 – ROC Curve of deep learning methods for OpenSSL.

ferent pairs, which increases the numbers of trained samples. When taking pairs as input, the model learns more useful features.

- The proposed VDSimilar performs the best among these methods. For example, it achieves 91.1% of F1-score, which is 2.5% and 6.2% higher than that of Siamese-Bilstm and Siamese-RNN respectively. Compared to Siamese-Bilstm, VDSimilar adds an attention layer so that the model focuses on the important snippets upon training.

5.2.2. *Performance in accuracy*

We further compare the performance of different deep learning based methods in terms of accuracy, which depicts an overall performance in vulnerability detection. Here, we report both the accuracy upon the training stage and the testing stage.

Fig. 6 compares the accuracy of different deep learning models for Linux data set. On the training set, SyseVR reaches 94.78% and VulDeePecker achieves 85.6% in terms of accuracy. However, on the testing set, these two methods perform not as well as on the training set. Especially for SyseVR, the accuracy is only 51.87%. The result for OpenSSL data set demonstrated in Fig. 7 show similar trends, i.e., Both SyseVR and VulDeePecker perform well on the training set yet poor on the testing set. This is mainly because the two models are over-fitted when training on a small data set.

For our proposed methods using the Siamese network, they takes pairs as input to enlarge the number of inputs, which helps solve the over-fitting problem when the amount of data is too small. As a result, these methods show well generalization on the testing set. As depicted in Figs. 6 and 7, for OpenSSL data set, the accuracy of VDSimilar is 93.62% in the training set and is 90.81% in the testing set. The difference between the two is only 2.81%. For Linux data set, the three value is 95.31%, 91.75% and 3.56% respectively. The improvement is mainly contributed to the Siamese network with Bilstm and Attention, so that VDSimilar learns the important features in the view of vulnerabilities and thus outperforms other methods in accuracy.

5.2.3. *Performance in ROC and AUC*

The ROC (2021) curve reflects the relationship between sensitivity and specificity. According to the position of the curve, the whole graph is divided into two parts. The area under the curve is called AUC (Area Under Curve), which represents the accuracy of prediction. The higher the AUC value, the larger area under the curve of ROC, and the higher the prediction accuracy. Fig. 8 depicts the ROC Curve of various deep learning methods for the OpenSSL data set. The area under the ROC curve of VDSimillar is the largest. It can be seen that when the false positive is close to 0, the true positive of VDSimilar is still higher than 0.85, implying that it can more accurately detect vulnerabilities.

Fig. 9 compares the AUC value of these methods on the testing set. As we can see, VDSimilar achieves the highest AUC value, i.e., it reaches up to 87.69% for the Linux testing set and 97.17% for the OpenSSL testing set. In addition, the AUC value of deep learning-based methods are higher than that of traditional methods. This is because the traditional methods such as Simian and ReDebug tend to compute similarity in the view of syntax and semantics, which cannot characterize the features of code in the view of vulnerability, thereby leading to false positives and false negatives.

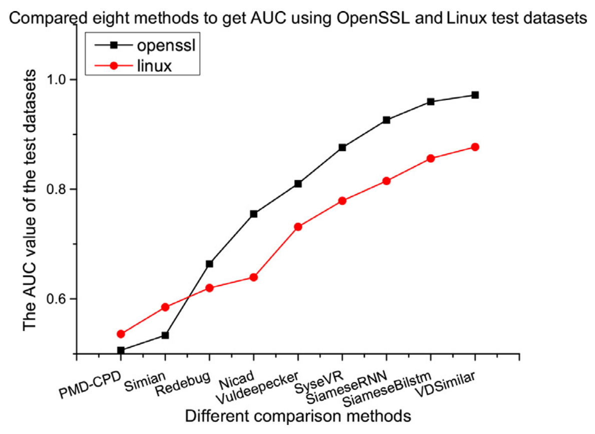


Fig. 9 – Comparison of AUC value.

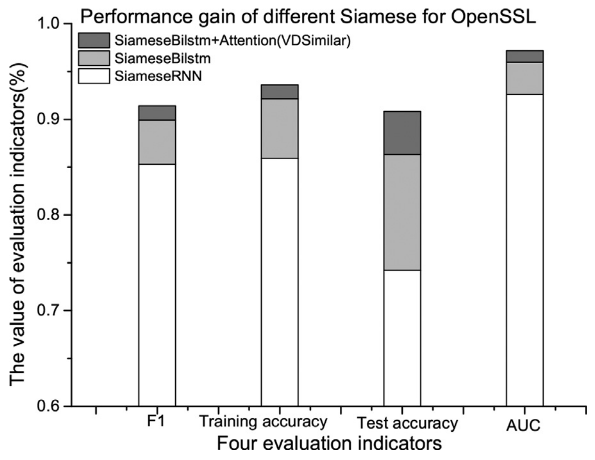


Fig. 10 – Performance gain of different efforts for OpenSSL.

5.3. *Performance gain of different efforts*

In addition to using Siamese network, we add RNN, BiLSTM and Attention to improve the performance. Here we measure the performance gained by these efforts in terms of F1-score, accuracy (on both the training set and testing set) and AUC. Figs. 10 and 11 demonstrate the performance gain of these efforts for OpenSSL and Linux respectively.

From Figs. 10 and 11, it can be seen that Siamese-BiLSTM brings about 4% improvement in AUC and 17% improvement in F1-score compared to Siamese-RNN. This is because BiLSTM can not only better capture the long-distance dependency between words, but also acquire the two-way semantic relationship. Thus, this is helpful to detect vulnerabilities when the snippets are scattered in code. The use of Attention brings about 2% of improvement over Siamese-BiLSTM in AUC, and 2.4% in F1-score. With the attention mechanism, VDSimilar can focus more on the important lines of code, thereby capturing the most important semantic information in vulnerable snippets.

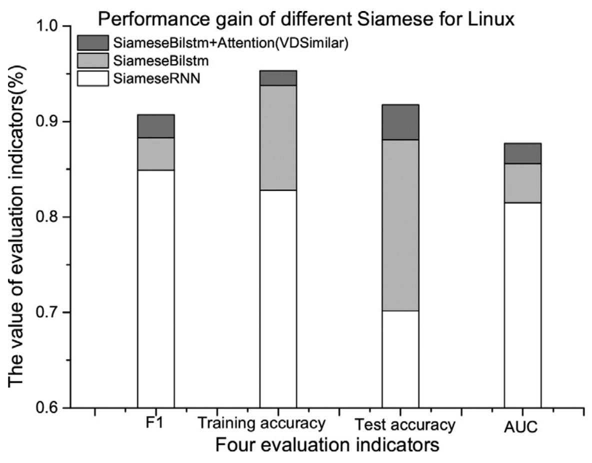


Fig. 11 – Performance gain of different efforts for Linux.

Table 9 – Performance with various settings for Linux.

Group	Settings	A(Train)(%)	A(Test)(%)
1	Attention Head = 1	94.38	90.95
	Attention Head = 2	95.31	91.75
	Attention Head = 4	94.97	91.23
2	Embedding Size = 32	91.35	87.10
	Embedding Size = 64	95.31	91.75
	Embedding Size = 128	95.51	89.82
3	Batch Size = 32	95.34	90.88
	Batch Size = 64	95.31	91.75
	Batch Size = 128	94.43	89.40

Table 10 – Performance with various settings for OpenSSL.

Group	Settings	A(Train)(%)	A(Test)(%)
1	Attention Head = 1	92.63	88.28
	Attention Head = 2	92.41	89.06
	Attention Head = 4	93.62	90.82
2	Embedding Size = 32	88.10	88.28
	Embedding Size = 64	93.62	90.82
	Embedding Size = 128	92.63	88.28
3	Batch Size = 32	89.74	90.63
	Batch Size = 64	93.62	90.82
	Batch Size = 128	92.58	90.71

5.4. *Determining the optimal model*

We mainly focus on three key variables in VDSimilar to choose the optimal model, Number of Heads in Attention, Embedding Size and Batch Size, as listed in Tables 9 and 10.

Attention head We first vary the number of attention heads in VDSimilar, i.e., 1, 2, and 4 respectively, and compare its performance on Linux and OpenSSL data set. As reported in Table 9, it can be seen that VDSimilar achieves the best performance when the attention head is 2. On the OpenSSL data set, as shown in Table 10, the accuracy keeps increasing with the increase of number of attention head. These results suggest that the use of Attention should adapt the data set to better capture the important vulnerability snippets.

Table 11 – Default settings of parameters.

Parameter	value
Epoch	150
Dropout	0.2
Batch size	64
Learning rate	0.001
Hidden layers in BiLSTM	2
Dimension of hidden units in BiLSTM	32
Dimension of the words embedding matrix W	64
Dimension of matrix in Attention W_Q, W_K, W_V	32

Table 12 – Time overhead on testing a function for Linux and OpenSSL (seconds).

	Linux	Openssl
ReDebug	0.068	0.068
Simian	0.221	0.236
Nicad	0.312	0.263
Sysevr	0.0041	0.0028
VulDeePecker	0.0042	0.0031
Siamese-RNN	0.0017	0.0077
Siamese-Bilstm	0.0035	0.0171
VDSimilar	0.0046	0.0191

In the following two groups of experiments, we by default set the attention head as 2 for the Linux data set and 4 for the OpenSSL data set. In addition, the other parameters are listed in Table 11.

Embedding size In terms of embedding size, it can be seen that VDSimilar performs the best when the embedding size is 64. E.g., the accuracy on the OpenSSL training set is 88.1% when the size is 31, and it achieves 93.62% when the size increases to 64. However, it then drops to 92.63% when the size is 128. This is because a smaller size embedding cannot well represent the hidden features, while a larger one may contain redundant features which hurts performance.

Batch size In terms of batch size, it can be seen that VDSimilar performs the best when the batch size is configured with 64. For example, the accuracy on the training set is 89.74% when the size is 32, reaches up to 93.62% when the size increases to 64, and then decreases to 92.58% when the size is 128. This is because the batch size denotes the number of samples for one epoch training. It is known that a small size leads to gradient oscillation and makes the model cannot converge, while a large size fails to find the gradient direction and thus falls into local optimal solution.

5.5. *Detection efficiency*

In this section, we compare the detection efficiency of VDSimilar against other methods. We measure the efficiency by the time overhead upon testing.

The time overhead for testing a function is directly related to the length of both vulnerability and testing function. Here, we report the average time of testing a function for OpenSSL and Linux.

As can be seen in Table 12, traditional tools (i.e., ReDebug, Simian and NiCad) take more time to test a function than deep learning-based tools. Because the traditional tools need to compare the target function with all vulnerability functions, so that the total comparison time is longer than prediction based methods. Note that ReDebug takes less time to test a function than Simian and NiCad, because it computes the hash value for sequence tokens and then simply finds matches by hash lookup.

The Siamese network with RNN and Bilstm take less time than SyseVR and VulDeePecker, because the Siamese structure uses a simple network architecture and thus is more efficient than the two deep learning models. VDSimilar, which extends Siamese-Bilstm with Attention layers, takes more time upon detection. E.g., on Linux set, it takes 0.0046 s yet Siamese-Bilstm takes 0.0035 s. The increase is acceptable compared to the gain in accuracy in vulnerability detection. These results show that VDSimilar is efficient in detecting vulnerabilities.

6. Discussion and limitation

Compared to existing methods that builds deep learning models on a large data set, we first prepare a relatively small data set. The data set contain a set of vulnerability functions and corresponding patched functions, which helps characterize vulnerability snippets. Then, we perform detection with Siamese network combined with BiLSTM and attention to learn similarity in the view of vulnerability snippets. The main advantage of our work over state-of-the-art lies in that it performs vulnerability detection with Siamese network on a much smaller data set, e.g., 876 functions compared to over 61,638 in Li et al. (2018b). Note that He et al. (2020) also intend to train a model on a small data set. However, they employ few-shot learning to measure the similarity between codes and cloned variants. In contrast, we propose to use Siamese network that trains on vulnerabilities and patches so that the model can learn the similarity in the view of vulnerability. In addition, the data set they used contain a lot of synthetic codes, i.e., Type-1, Type-2 and Type-3 variants. In contrast, our data set record vulnerabilities and patches collected from real-world programs.

There still exist several limitations of the proposed method. For example, the count of pairs for many CVEs is still not enough for training, there exist many security irrelevant information in the code that may hurt the performance of the detection model. In addition, although the proposed method performs well in detecting vulnerabilities, why it works well, i.e, interpretability of the proposed model, is still unknown. In the future, we will work towards these directions.

7. Conclusions

In this paper, we present a code similarity based vulnerability detection approach. Compared to current work that training a classifier on vulnerabilities, VDSimilar is trained on pairs of vulnerabilities and patches with the aim to compute similarity between two pieces of codes. Therefore, it can better describe similar code for vulnerabilities. In addition, with a metric learning model, our approach can detect newly emerged vulnerabilities without retraining. The experimental results show that our VDSimilar is effective in detecting vulnerabilities.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

CRediT authorship contribution statement

Hao Sun: Conceptualization, Methodology, Resources, Writing – original draft. **Lei Cui:** Investigation, Resources, Writing – review & editing, Supervision. **Lun Li:** Investigation. **Zhenquan Ding:** Supervision. **Zhiyu Hao:** Writing – review & editing, Supervision. **Jiancong Cui:** Resources. **Peng Liu:** Investigation.

Acknowledgments

This work was supported in part by the National Natural Science Foundation of China (Grant nos. 61972392, 62072453) and Youth Innovation Promotion Association of the Chinese Academy of Sciences (Grant no. 2020164).

REFERENCES

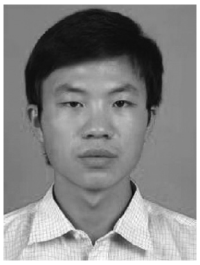
Chakraborty S, Krishna R, Ding Y, Ray B. Deep learning based vulnerability detection: are we there yet? IEEE Trans. Softw. Eng. 2020.
Chopra S, Hadsell R, LeCun Y. Learning a similarity metric discriminatively, with application to face verification, 1. IEEE; 2005. p. 539–46.
Feng H, Fu X, Sun H, Wang H, Zhang Y. Efficient vulnerability detection based on abstract syntax tree and deep learning. In: IEEE INFOCOM 2020-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS); 2020. p. 722–7.
Graves A, Jaitly N, Mohamed A-r. Hybrid speech recognition with deep bidirectional LSTM. In: 2013 IEEE Workshop on Automatic Speech Recognition and Understanding. IEEE; 2013. p. 273–8.
Graves A, Schmidhuber J. Framewise phoneme classification with bidirectional LSTM and other neural network architectures. Neural Netw. 2005;18(5–6):602–10.
Grieco G, Grinblat GL, Uzal L, Rawat S, Feist J, Mounier L. Toward large-scale vulnerability discovery using machine learning. In: Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy; 2016. p. 85–96.
He Y, Wang W, Sun H, Zhang Y. Vul-mirror: a few-shot learning method for discovering vulnerable code clone. EAI Endorsed Trans. Secur. Saf. 2020;7(23):e4.
Hochreiter S, Schmidhuber J. Long short-term memory. Neural Comput. 1997;9(8):1735–80.
Hunt JW, MacIlroy MD. An Algorithm for Differential File Comparison. Bell Laboratories Murray Hill; 1976.
Jang J, Agrawal A, Brumley D. ReDeBug: finding unpatched code clones in entire os distributions. In: 2012 IEEE Symposium on Security and Privacy. IEEE; 2012. p. 48–62.
Jiang L, Misherghi G, Su Z, Glondu S. DECKARD: scalable and accurate tree-based detection of code clones. In: 29th International Conference on Software Engineering (ICSE'07). IEEE; 2007. p. 96–105.
Kamiya T, Kusumoto S, Inoue K. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. IEEE Trans. Softw. Eng. 2002;28(7):654–70.
Kim S, Woo S, Lee H, Oh H. VUDDY: a scalable approach for vulnerable code clone discovery. In: 2017 IEEE Symposium on Security and Privacy (SP). IEEE; 2017. p. 595–614.
Li, Y., Gu, C., Dullien, T., Vinyals, O., Kohli, P., 2019. Graph matching networks for learning the similarity of graph structured objects. arXiv preprint arXiv:1904.12787
Li, Z., Zou, D., Xu, S., Chen, Z., Zhu, Y., Jin, H., 2020. Vuldeelocator: a deep learning-based fine-grained vulnerability detector.
Li Z, Zou D, Xu S, Jin H, Qi H, Hu J. VulPecker: an automated vulnerability detection system based on code similarity analysis. In: Proceedings of the 32nd Annual Conference on Computer Security Applications; 2016. p. 201–13.
Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y., Chen, Z., 2018a. SySeVR: a framework for using deep learning to detect software vulnerabilities. arXiv:1807.06756
Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., Zhong, Y., 2018b. VulDeePecker: a deep learning-based system for vulnerability detection. arXiv preprint arXiv:1801.01681
Liang H, Xie Z, Chen Y, Ning H, Wang J. FIT: inspect vulnerabilities in cross-architecture firmware by deep learning and bipartite matching. Comput. Secur. 2020;99:102032.
Mazuera-Rozo, A., Mojica-Hanke, A., Linares-Vásquez, M., Bavota, G., 2021. Shallow or deep? An empirical study on detecting vulnerabilities using deep learning. arXiv preprint arXiv:2103.11940
Neculoiu P, Versteegh M, Rotaru M. Learning text similarity with siamese recurrent networks. In: Proceedings of the 1st Workshop on Representation Learning for NLP; 2016. p. 148–57.
NVD and SARD, https://github.com/SySeVR/SySeVR.
Pendleton M, Garcia-Lebron R, Cho J-H, Xu S. A survey on systems security metrics. ACM Comput. Surv. (CSUR) 2016;49(4):1–35.
PMD-CPD, is abbreviated as CPD. https://en.wikipedia.org/wiki/PMD_(software).
Roy CK, Cordy JR. NICAD: accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In: 2008 16th iEEE International Conference on Program Comprehension. IEEE; 2008. p. 172–81.
ROC, https://en.wikipedia.org/wiki/Receiver_operating_characteristic.
Russell R, Kim L, Hamilton L, Lazovich T, Harer J, Ozdemir O, Ellingwood P, McConley M. Automated vulnerability detection in source code using deep representation learning. In: 2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA). IEEE; 2018. p. 757–62.
Siamese-RNN and Siamese-BiLSTM, https://github.com/ascourge21/Siamese.
Simian, http://www.harukizaemon.com/simian/.
Su J, Tan Z, Xiong D, Ji R, Shi X, Liu Y. Lattice-based recurrent neural network encoders for neural machine translation. Thirty-First AAAI Conference on Artificial Intelligence, 2017.
Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser Ł, Polosukhin I. Attention is all you need. In: Advances in Neural Information Processing Systems; 2017. p. 5998–6008.
Vinyals O, Toshev A, Bengio S, Erhan D. Show and tell: a neural image caption generator. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition; 2015. p. 3156–64.
Wang H, Ye G, Tang Z, Tan SH, Huang S, Fang D, Feng Y, Bian L, Wang Z. Combining graph-based learning with automated

data collection for code vulnerability detection. IEEE Trans. Inf. Forensics Secur. 2020.
Xu X, Liu C, Feng Q, Yin H, Song L, Song D. Neural network-based graph embedding for cross-platform binary code similarity detection. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security; 2017. p. 363–76.
Zou D, Wang S, Xu S, Li Z, Jin H. µVulDeePecker: a deep learning-based system for multiclass vulnerability detection. IEEE Trans. Dependable Secure Comput. 2019:1.

Hao Sun received the B.S. degree in software engineering from Harbin University of Science and Technology in 2019. She is currently pursing the Ph.D. degree with the School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China. Her research interests include network security, malicious code detection and deep learning.

Lei Cui is an associate professor of Institute of Information Engineering, Chinese Academy of Sciences. He received his Doctor's degree in Computer Software and Theory from Beihang University in 2015. His research interests include operating system, distributed systems and system virtualization. He has published over 20 papers in journals and conferences including VEE, LISA, DSN, The Computer Journal, TPDS.

Lun Li is currently a senior engineer of Institute of Information Engineering, Chinese Academy of Sciences. She received her doctor's degree in Information Security from University of Chinese Academy of Sciences in 2019. Her research interests include network security, system virtualization and network emulation. She has published over 6 papers in journals and conferences including ICA3PP, ICPADS, HPCC.


Zhenquan Ding is currently a associate professor of Institute of Information Engineering, Chinese Academy of Sciences. He received his Master's degree in Computer Science and Technology from Harbin Institute of Technology in 2012. His research interests include cyberspace security, system virtualization and network emulation. He has published many papers in conferences including ICA3PP, HPCC and PDCAT. He has applied for over 20 invention patents and granted 9 patents.

Zhiyu Hao is currently a professor of Institute of Information Engineering, Chinese Academy of Sciences. He received his Doctor's degree in Computer System Architecture from Harbin Institute of Technology in 2007. His research interests include network security, system virtualization and network emulation. He has published over 30 papers in journals and conferences including ICPP, IEEE S&P, ICA3PP and CLUSTER.

Jiancong Cui is currently a guest student of Institute of Information Engineering, Chinese Academy of Sciences. He is an undergraduate of Shandong Normal University (2017-). His research interest is in the area of machine learning and network security.

Peng Liu received his Ph.D. degree from the Beihang University, China in 2017. He joined the Guangxi Normal University as an assistant professor in 2007. Since 2015, he has been an associate professor. His current research interests include network security, data privacy, and graph mining.