VulCNN: An Image-inspired Scalable Vulnerability Detection System

Yueming Wu[*][†]
Huazhong University of Science and Technology, China
wuyueming@hust.edu.cn

Deqing Zou[*][†][‡]
Huazhong University of Science and Technology, China
deqingzou@hust.edu.cn

Shihan Dou
Fudan University, China
shihandou@foxmail.com

Wei Yang
University of Texas at Dallas, United States
wei.yang@utdallas.edu

Duo Xu[*]
Huazhong University of Science and Technology, China
u201714569@hust.edu.cn

Hai Jin[†][§]
Huazhong University of Science and Technology, China
hjin@hust.edu.cn

ABSTRACT

Since *deep learning* (DL) can automatically learn features from source code, it has been widely used to detect source code vulnerability. To achieve scalable vulnerability scanning, some prior studies intend to process the source code directly by treating them as text. To achieve accurate vulnerability detection, other approaches consider distilling the program semantics into graph representations and using them to detect vulnerability. In practice, text-based techniques are scalable but not accurate due to the lack of program semantics. Graph-based methods are accurate but not scalable since graph analysis is typically time-consuming.

In this paper, we aim to achieve both scalability and accuracy on scanning large-scale source code vulnerabilities. Inspired by existing DL-based image classification which has the ability to analyze millions of images accurately, we prefer to use these techniques to accomplish our purpose. Specifically, we propose a novel idea that can efficiently convert the source code of a function into an image while preserving the program details. We implement *VulCNN* and evaluate it on a dataset of 13,687 vulnerable functions and 26,970 non-vulnerable functions. Experimental results report that *VulCNN* can achieve better accuracy than eight state-of-the-art vulnerability detectors (*i.e., Checkmarx*, *FlawFinder*, *RATS*, *TokenCNN*, *VulDeePecker*, *SySeVR*, *VulDeeLocator*, and *Devign*). As for scalability, *VulCNN* is about four times faster than *VulDeePecker* and *SySeVR*, about 15 times faster than *VulDeeLocator*, and about six times faster than *Devign*. Furthermore, we conduct a case study on more than 25 million lines of code and the result indicates that *VulCNN* can detect large-scale vulnerability. Through the scanning reports, we finally discover 73 vulnerabilities that are not reported in NVD.

[*]Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, HUST, Wuhan, 430074, China
[†]National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, HUST, Wuhan, 430074, China
[‡]Corresponding author
[§]Cluster and Grid Computing Lab, School of Computer Science and Technology, HUST, Wuhan, 430074, China

CCS Concepts

• **Security and privacy → Vulnerability scanners**.

Keywords

Vulnerability Detection, CNN, Large Scale, Image

ACM Reference Format:
Yueming Wu, Deqing Zou, Shihan Dou, Wei Yang, Duo Xu, and Hai Jin. 2022. VulCNN: An Image-inspired Scalable Vulnerability Detection System. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3510003.3510229

1 INTRODUCTION

Recently, various cyberspace security incidents [2, 3] such as hacker extortion, botnet attacks, and user information leakage have occurred frequently. As an important part of cyberspace, vulnerabilities in software systems have brought serious security threats to cyberspace. In 2020, Synopsys *Open Source Security and Risk Analysis* (OSSRA) examined audit data from 1,250+ commercial codebases and revealed that 70% of the code in these codebases were open-source. Moreover, 75% of these open-source codebases contained open source security vulnerabilities, and nearly half contained high-risk vulnerabilities [1]. Therefore, it is urgent to carry out large-scale and intelligent software vulnerabilities detection methods to better protect software security.

In general, source code vulnerability detection methods can be divided into two main categories, that is, the code-similarity-based methods [27, 32, 37, 39, 46] and the pattern-based methods [5, 6, 12, 24, 41, 43, 47, 56, 57]. Vulnerability detection methods based on code similarity are mainly used to detect vulnerabilities caused by code cloning. When used to detect vulnerabilities not caused by code cloning, it will lead to a high false negative rate [41]. Traditional pattern-based vulnerability detection methods [5, 6, 12] rely on experts to manually define vulnerability rules or characteristics to describe vulnerabilities. These approaches are not only subjective but also difficult to achieve a low false positive rate and a low false negative rate at the same time [40, 41].

In recent years, due to the automatic feature extraction of *deep learning* (DL), it has been widely used to detect source code vulnerability. These DL-based techniques [24, 40, 41, 43, 47, 56, 57] belong to the second category of methods (*i.e.,* pattern-based methods). They do not require experts to manually define features and can automatically generate vulnerability patterns. For example, some prior studies [41, 47] treat the program source code as text and apply techniques in the field of natural language processing to detect vulnerability. The detection performance of these text-based methods is not ideal since they ignore the program semantics of source code. To address the issue, researchers conduct program analysis to distill the program semantics of source code into a graph representation and perform graph analysis (*e.g.,* graph neural network) to detect vulnerability. These graph-based techniques [24, 56] can achieve higher effectiveness on detecting vulnerability, however, their scalability is much worse than text-based methods. In addition, almost all these methods only focus on labeling a function as either vulnerable or not, and cannot pinpoint which lines of code may be more likely to be vulnerable.

In this paper, we aim to achieve accuracy and scalability simultaneously on detecting vulnerabilities from large-scale source code. Our key idea is derived from DL-based image classification, which can process millions of images while maintaining high accuracy, and the classification results can be interpreted by visualization techniques. Specifically, we mainly address one major challenge in our paper.

- *How to efficiently convert the source code of a function into an image while preserving the program details?*

To tackle the challenge, we first conduct program analysis to distill the program semantics of a function into a *program dependency graph* (PDG) which contains both control-flow and data-flow details of source code. After obtaining the PDG of a function, we treat it as a social network and apply centrality analysis on the network to attach the graph structural information to each line of code. In social network analysis, centrality analysis [25, 31] has been proposed to measure the importance of a node within the network. Specifically, we leverage three different centralities (*i.e.,* degree centrality [25], katz centrality [31], and closeness centrality [25]) to commence our image transformation. There are two main reasons for using three centralities. First, different centralities can maintain the graph properties from different aspects [52]. Second, an image generally has three channels (*i.e.,* red, green, and blue) and they work together to produce a complete image. The output of centrality analysis is an image while preserving the graph details from three aspects. Given generated images, we then train a *Convolutional Neural Network* (CNN) [34, 36] model and use it to detect vulnerability. To pinpoint the vulnerable lines of code in a function, we use a deep visualization technique (*i.e.,* Class Activation Map [20]) on our images to obtain the corresponding heatmaps, these heatmaps can help security analysts understand why the function is labeled as a vulnerability.

We implement *VulCNN* and evaluate it on a dataset of 40,657 functions which consists of 13,687 vulnerable functions and 26,970 non-vulnerable functions. Evaluation results show that *VulCNN* can achieve better effectiveness than eight comparative vulnerability detectors (*i.e., Checkmarx* [5], *FlawFinder* [6], *RATS* [12], *TokenCNN* [47], *VulDeePecker* [41], *SySeVR* [40], *VulDeeLocator* [38], and *Devign* [56]). Furthermore, *VulCNN* is more than six times faster than another state-of-the-art graph-based vulnerability detection tool (*i.e., Devign*). To validate the ability of *VulCNN* on large-scale vulnerability scanning, we conduct a case study on more than 25 million lines of code. Through the scanning reports, we finally discover 73 vulnerabilities that are not reported in NVD. Among them, 17 have been "silently" patched by vendors in the latest version of corresponding products, four vulnerabilities have been deleted, and the other 52 still exist in the products. We have reported these vulnerabilities to their vendors and hope that they can be patched as soon as possible.

In summary, this paper makes the following contributions:

- We propose a novel idea that can efficiently convert the source code of a function into an image while preserving the program details.
- We design and implement a prototype system, *VulCNN*[1], a scalable graph-based vulnerability detection system.
- We conduct evaluations on a dataset of 13,687 vulnerable functions and 26,970 non-vulnerable functions. Experimental results report that *VulCNN* is superior to eight state-of-the-art vulnerability detectors (*i.e., Checkmarx*, *FlawFinder*, *RATS*, *TokenCNN*, *VulDeePecker*, *SySeVR*, *VulDeeLocator*, and *Devign*).
- We conduct a case study on more than 25 million lines of code to validate the ability of *VulCNN* on large-scale vulnerability scanning. Through the scanning results, we discover 73 vulnerabilities that are not reported in NVD.

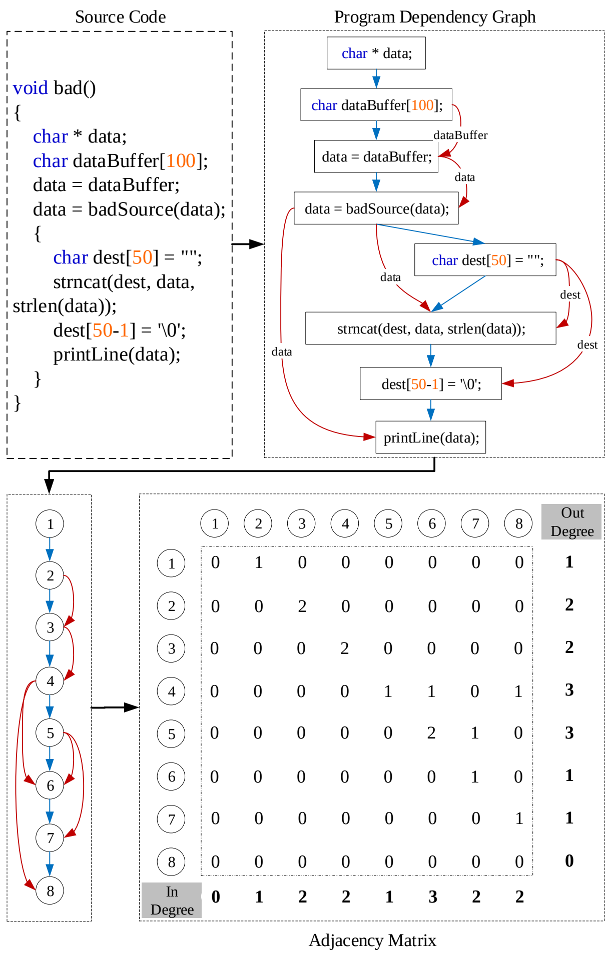
Paper organization. The remainder of the paper is organized as follows. Section 2 presents the motivation of our paper. Section 3 introduces our system. Section 4 reports the experimental results. Section 5 discusses the future work. Section 6 describes the related work. Section 7 concludes the present paper.

2 MOTIVATION

We all know that a function consists of multiple lines of code, which together implement the program semantics (*i.e.,* functionality) of the function. However, the semantic contribution of different lines of code is different. For example, some code is just a simple variable definition, while some code implements the core algorithm of the function. Obviously, the latter contributes more semantics. Therefore, we raise a question: *"How to find out the contribution of different lines of code in the function to the program semantics?"*

To answer it, we select a buffer overflow vulnerability function as our example. To maintain the program semantics, we consider extracting both control-flow and data-flow details by static analysis. Specifically, we leverage *Joern* [53] to obtain the *program dependency graph* (PDG) of the vulnerability in Figure 1. Each node in the PDG corresponds to a line of code in the vulnerability. Red lines and blue lines show the data flows and control flows between different lines of code in the function, respectively. To simply display the PDG of the vulnerability, we replace each line of code with a numbered circular node. Eight lines of code in the function correspond to eight circular nodes as shown in Figure 1. In graph theory, an adjacency matrix is a square matrix used to

[1]https://github.com/CGCL-codes/VulCNN



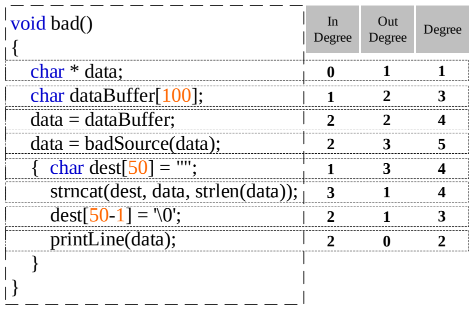
```
void bad()
{
    char * data;
    char dataBuffer[100];
    data = dataBuffer;
    data = badSource(data);
    {
        char dest[50] = "";
        strncat(dest, data,
strlen(data));
        dest[50-1] = '\0';
        printLine(data);
    }
}
```

	1	2	3	4	5	6	7	8	Out Degree
1	0	1	0	0	0	0	0	0	**1**
2	0	0	2	0	0	0	0	0	**2**
3	0	0	0	2	0	0	0	0	**2**
4	0	0	0	0	1	1	0	1	**3**
5	0	0	0	0	0	2	1	0	**3**
6	0	0	0	0	0	0	1	0	**1**
7	0	0	0	0	0	0	0	1	**1**
8	0	0	0	0	0	0	0	0	**0**
In Degree	**0**	**1**	**2**	**2**	**1**	**3**	**2**	**2**	

Figure 1: The *program dependency graph* (PDG) and corresponding adjacency matrix of a vulnerability

represent a finite graph [4]. Therefore, to represent the PDG of the vulnerability, we compute the corresponding adjacency matrix and describe it in Figure 1. The elements of the matrix indicate whether pairs of nodes are adjacent or not in the graph. Since our PDG is a directed graph, an element represents the number of directed edges between two different nodes. For example, node 3 has two edges pointing to node 4, while node 4 has no edges pointing to node 3. Therefore, the corresponding values between node 3 and node 4 in the matrix are two and zero, respectively.

After obtaining the adjacency matrix of the PDG, we find that when we sum a row in the matrix, the value obtained corresponds to the out-degree of the node. Moreover, when we sum a column, the value obtained is the in-degree of the node. For example, the sum of the row of node 4 is three (*i.e.,* 1+1+1=3), which indicates that the out-degree of node 4 is three. To better illustrate our insight, we present the in-degrees, out-degrees, and degrees of all lines of code of the vulnerability in Figure 2. Through the results in Figure 2, we find that the degrees of different lines of code are basically different. It is reasonable because there are different relationships



Code	In Degree	Out Degree	Degree
void bad()			
{			
char * data;	**0**	**1**	**1**
char dataBuffer[100];	**1**	**2**	**3**
data = dataBuffer;	**2**	**2**	**4**
data = badSource(data);	**2**	**3**	**5**
{ char dest[50] = "";	**1**	**3**	**4**
strncat(dest, data, strlen(data));	**3**	**1**	**4**
dest[50-1] = '\0';	**2**	**1**	**3**
printLine(data);	**2**	**0**	**2**
}			
}			

Figure 2: The in-degrees, out-degrees, and degrees of all lines of code in Figure 1

(*i.e.,* control-flows and data-flows) between different lines of code, and the vulnerability is triggered based on these relationships. If we directly process these codes by treating them as text, the degrees of all lines of code are one, which may decrease the vulnerability detection accuracy.

In one word, a graph can be represented by its adjacency matrix, the matrix can be described by degrees of all nodes. Therefore, computing the degrees of code in a function may be a great candidate to retain the graph details. In practice, the degree of a node in a graph is originally used to quantify its importance. It has been widely used in social network analysis [25]. The higher the degree, the more important the person. Meanwhile, different lines of code have different degrees. If we treat each line of code as a person, the control-flow and data-flow relationships as the communications between persons, then the corresponding PDG can be regarded as a social network. The higher the degree of a person, the more other persons communicated with him, and the greater his importance within the PDG social network. Therefore, we can leverage the importance of a line of code as a form of the contribution of program semantics. In other words, the more important a line of code is, the more it may contribute to implementing the program semantics (*i.e.,* functionality) of the function. Based on the observation, we design *VulCNN* by analyzing the importance of all lines of code.

3 SYSTEM

In this section, we introduce *VulCNN*, a novel and efficient source code vulnerability detection system.

3.1 Overview

As shown in Figure 3, *VulCNN* consists of four main phases: *Graph Extraction*, *Sentence Embedding*, *Image Generation*, and *Classification*.

- ***Graph Extraction:*** Given the source code of a function, we first normalize them and then perform static analysis to extract the program dependency graph of the function.
- ***Sentence Embedding:*** Each node in the program dependency graph corresponds to a line of code in the function. We regard a line of code as a sentence and embed them into a vector.
- ***Image Generation:*** After sentence embedding, we apply centrality analysis to obtain the importance of all lines of

code and multiply them by the vectors one by one. The output of this phase is an image.

- ***Classification:*** Our final phase focuses on classification. Given generated images, we first train a CNN model and then use it to detect vulnerability.

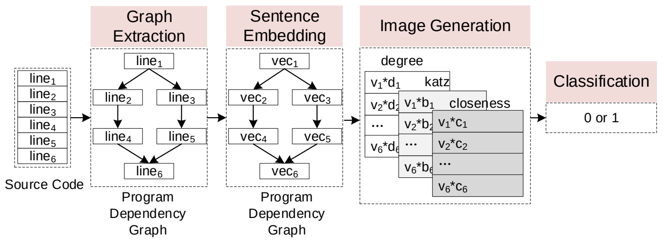
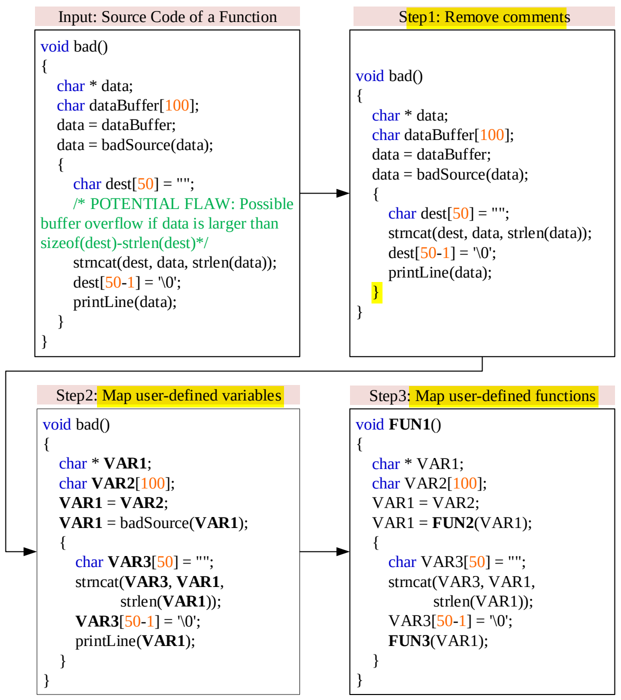


Figure 3: System overview of *VulCNN*

3.2 Graph Extraction and Sentence Embedding

VulCNN aims to achieve accuracy and scalability simultaneously on detecting vulnerabilities, therefore, we first apply static analysis to distill the program semantics of source code into a graph representation. Due to the coarse granularity of file-level vulnerability detection, we focus on detecting vulnerability at a more fine-grained level (*i.e.,* function-level) since a function can also implement a specific task. Before extracting the graph representation of a function, we first abstract and normalize the source code. Particularly, we use three levels of normalization which make *VulCNN* resilient to common code modifications while preserving program semantics.



Input: Source Code of a Function

```
void bad()
{
   char * data;
   char dataBuffer[100];
   data = dataBuffer;
   data = badSource(data);
   {
      char dest[50] = "";
      /* POTENTIAL FLAW: Possible
buffer overflow if data is larger than
sizeof(dest)-strlen(dest)*/
      strncat(dest, data, strlen(data));
      dest[50-1] = '\0';
      printLine(data);
   }
}
```

Step1: Remove comments

```
void bad()
{
   char * data;
   char dataBuffer[100];
   data = dataBuffer;
   data = badSource(data);
   {
      char dest[50] = "";
      strncat(dest, data, strlen(data));
      dest[50-1] = '\0';
      printLine(data);
   }
}
```

Step2: Map user-defined variables

```
void bad()
{
   char * VAR1;
   char VAR2[100];
   VAR1 = VAR2;
   VAR1 = badSource(VAR1);
   {
      char VAR3[50] = "";
      strncat(VAR3, VAR1,
            strlen(VAR1));
      VAR3[50-1] = '\0';
      printLine(VAR1);
   }
}
```

Step3: Map user-defined functions

```
void FUN1()
{
   char * VAR1;
   char VAR2[100];
   VAR1 = VAR2;
   VAR1 = FUN2(VAR1);
   {
      char VAR3[50] = "";
      strncat(VAR3, VAR1,
            strlen(VAR1));
      VAR3[50-1] = '\0';
      FUN3(VAR1);
   }
}
```

Figure 4: Steps of source code normalization in *VulCNN*

Figure 4 shows the detailed transformations of a function at varying normalization levels.

- ***Step1*:** Remove the comments in the source code because they have nothing to do with program semantics.
- ***Step2*:** Map user-defined variables to symbolic names in a one-to-one manner (*e.g.,* , VAR1).
- ***Step3*:** Map user-defined functions to symbolic names in a one-to-one manner (*e.g.,* , FUN1).

After abstracting the source code, we then leverage an open-source code analysis platform for C/C++, *Joern* [11, 53], to extract the *program dependency graph* (PDG) of the function. PDG is a graph representation that contains both data-flow and control-flow details of the source code. Each node in a PDG corresponds to a line of code in the function. We treat a line of code as a sentence and apply sentence embedding to transform it into a fixed-length vector. Specifically, we make use of a widely used method (*i.e.,* sent2vec [45]) to complete our sentence embedding. It adopts a simple but efficient unsupervised objective to train distributed representations of sentences. Using the sent2vec [45] model, we can transform a line of code into its corresponding vector representation whose dimension is 128 in our paper.

To better illustrate the detailed steps involved in our proposed method, we provide an example in Figure 5. Red lines and blue lines in Figure 5 represent data flows and control flows between different lines of code in a function, respectively.

3.3 Image Generation

After graph extraction and sentence embedding, we can obtain a new PDG where each node is a vector representation. In this phase, we aim to efficiently convert the new PDG into an image while considering the contributions of different lines of code to program semantics. To complete our purpose, we regard the new PDG as a social network and apply social network centrality analysis to obtain the importance of all lines of code. Centrality concepts were first developed in social network analysis whose original purpose is to measure the importance of a node in the network. Centrality analysis has been used in many different areas (*e.g.,* biological network [28] and transportation network [26]), these successful applications have validated the effectiveness of network analysis. In fact, there have been proposed many different types of centrality to quantify the importance of a node in a network from different aspects, for example:

- ***Degree centrality*** [25] of a node is the fraction of nodes it is connected to. The degree centrality values are normalized by dividing by the maximum possible degree in a graph N-1 where N is the number of nodes in the graph.

$$x_i = \frac{deg(i)}{N-1} \tag{1}$$

  Note that $deg(i)$ is the degree of node i.

- ***Katz centrality*** [31] computes the centrality for a node based on the centrality of its neighbors. The *Katz centrality* for node i is

$$x_i = \alpha \sum_j A_{ij} x_j + \beta \tag{2}$$

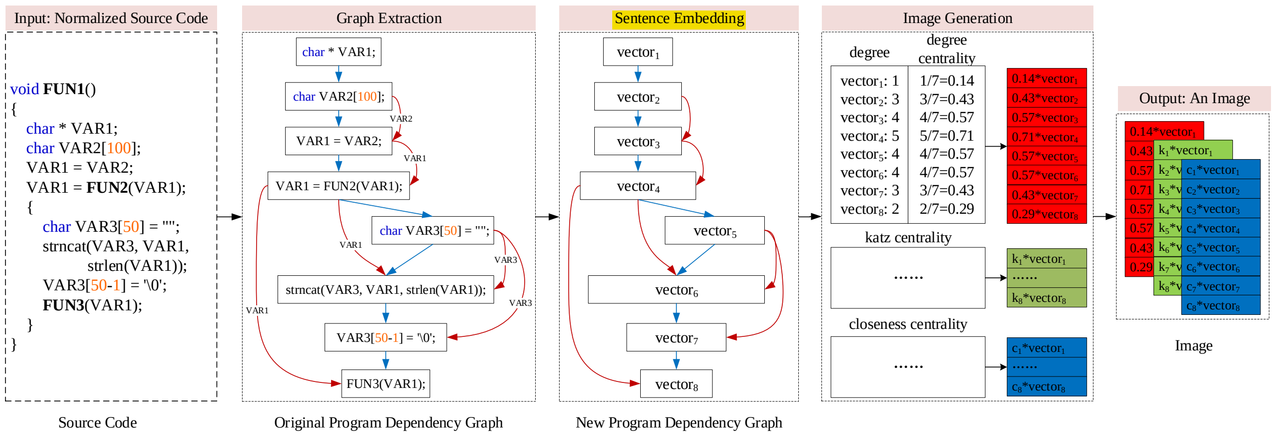


Figure 5: An example to illustrate how to convert the source code of a function into an image

Note that A is the adjacency matrix of the graph G with eigenvalues λ. The parameter β controls the initial centrality and

$$\alpha < \frac{1}{\lambda_{max}} \tag{3}$$

Katz centrality computes the relative influence of a node within a graph by measuring the number of the immediate neighbors (first-degree nodes) and also all other nodes in the graph that connect to the node under consideration through these immediate neighbors.

- ***Closeness centrality*** [25] indicates how close a node is to all other nodes in the network. It is calculated as the average of the shortest path length from the node to every other node in the graph. The smaller the average shortest distance of a node, the greater the closeness centrality of the node.

$$x_i = \frac{N-1}{\sum_{i \neq j} d(i,j)} \tag{4}$$

Note that $d(i,j)$ is the distance between nodes i and j, and N is the number of nodes in the graph.

Because an image generally has three channels (*i.e.,* red, green, and blue) and they work together to produce a complete image. We select three different centralities (*i.e.,* *degree centrality* [25], *katz centrality* [31], and *closeness centrality* [25]) to correspond to the three channels. These three centralities can compute the importance of all lines of code in a function from three different aspects. By this, we can achieve complete consideration of different lines of code's contribution to a function's program semantics. Algorithm 1 shows the entire process of *VulCNN* of how to convert a function into an image. As shown in Figure 5 and Algorithm 1, we first perform degree centrality analysis on all nodes in the new PDG to collect the degree centralities of all nodes (*i.e.,* vectors). All vectors are then arranged one by one according to the number of lines of code after multiplying by the corresponding degree centrality. We call these arranged new vectors the "degree channel". Similarly, after applying katz centrality and closeness centrality analysis on the new PDG, we can obtain the other two "channels" which are "katz channel" and "closeness channel". Finally, these three channels are used to produce an image.

Algorithm 1 Converting the source code of a function into an image

```
Input: F: Source code of a function;
Output: I: An image.
1: nF ← CodeNormalization(F)
2: PDG ← GraphExtraction(nF)
3: V ← ObtainNodes(PDG)
4: for each $v \in V$ do
5:     vector ← SentenceEmbedding(v)
6:     degreeCentrality ← DegreeCentralityAnalysis(vector, PDG)
7:     degreeChannel.add(degreeCentrality $*$ vector)
8:     katzCentrality ← KatzCentralityAnalysis(vector, PDG)
9:     katzChannel.add(katzCentrality $*$ vector)
10:    closenessCentrality ← ClosenessCentralityAnalysis(vector, PDG)
11:    closenessChannel.add(closenessCentrality $*$ vector)
12: end for
13: I ← ImageGeneration(degreeChannel, katzChannel, closenessChannel)
14: return I
```

In brief, the input of image generation phase is a new PDG, where each node is an embedded vector, and the output is an image with the importances of all lines of code.

3.4 Classification

Deep learning [35] is a complex machine learning algorithm that has achieved results in many areas (*e.g.,* speech and image recognition) far beyond previous related technologies. The advantage of deep learning is to use unsupervised or semi-supervised feature learning and efficient hierarchical feature extraction algorithms to replace manual feature acquisition. In the field of image processing, *Convolutional Neural Network* (CNN) [34, 36] has been the focus since it not only does not require manual image preprocessing but also can use its unique fine-grained feature extraction to reach a level close to humans.

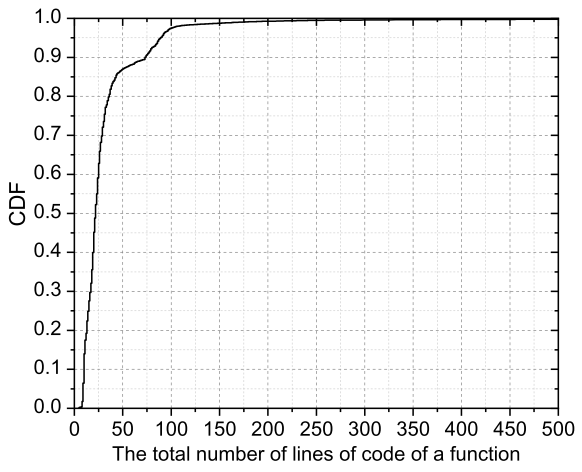


Figure 6: The *Cumulative Distribution Function* (CDF) of the total number of lines of code in a function

After image generation phase, the source code of a function is converted into an image. Given an image, we intend to train a CNN model first and then use it to detect vulnerability. Since CNN takes equal-size images as input while the number of lines of code in different functions is different, we need to make an adjustment. To find a more suitable threshold to produce fixed-size images, we select our experimental dataset in Section 4 (*i.e.,* 40,657 functions) as the test object and record the number of lines of code of all these functions. Figure 6 shows the *Cumulative Distribution Function* (CDF) of the number of lines of code in these functions. From the results in Figure 6, it can be observed that more than 99% of the functions have less than 200 lines of code. In reality, we have experimented with different thresholds (*i.e.,* 50-200 lines of code) on detecting vulnerability. Considering the detection accuracy and corresponding runtime overhead[2], we finally choose 100 lines of code as the threshold to generate our input images. When the number of lines of code in a function is less than 100, we pad zeros to the end of the vectors. When a function has more than 100 lines of code, we delete the ending part of vectors. The size of the image we input is 3∗100∗128, where 3 corresponds to three channels (*i.e.,* "degree channel", "katz channel", and "closeness channel"), 100 corresponds to the threshold of code lines, and 128 represents the dimension of a sentence vector.

Table 1: Parameter settings in *VulCNN*

parameters	settings
loss function	Cross Entropy Loss
activation function	ReLU
optimizer	Adam
batch size	32
learning rate	0.001
epoch num	100

[2]More details are in Section 4.2.

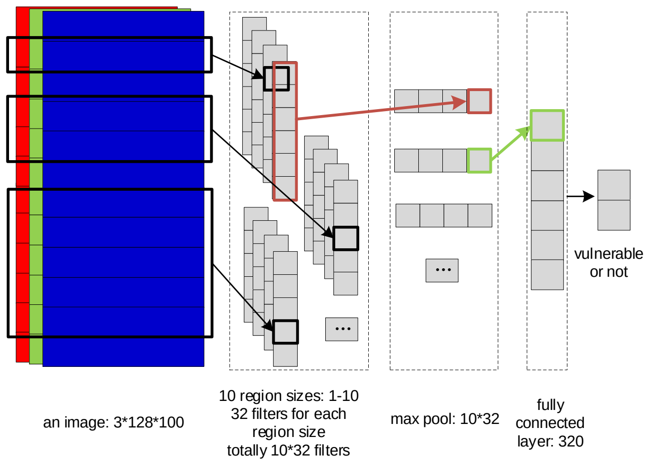


Figure 7: CNN classification of *VulCNN*

After generating fixed-size images, we then train a CNN model using these images. As shown in Figure 7, we use different convolution filters with a shape of m∗128, so that each filter can span the entire space of the sentence embedding. The filter size m determines the number of sequential sentences considered together. In *VulCNN*, we select 10 filter sizes (*i.e.,* 1 to 10), and each size has 32 feature maps to extract the features of different parts of the image. After max pooling, the length of our fully connected layer is 320. The descriptions of parameters used in *VulCNN* are illustrated in Table 1. The whole model uses rectified linear units (*i.e.,* ReLU [23]) as the non-linear activation function. Moreover, the loss function used in our CNN to penalize the incorrect classification is cross-entropy loss. We train the CNN using Adam [33] with a learning rate of 0.001. After obtaining a trained CNN model, we then use it to flag new functions as either vulnerable or not.

To interpret the detection results, we leverage *Gradient-weighted Class Activation Mapping++* (Grad-CAM++) [20, 49] as our visualization technique to pinpoint the vulnerable lines of code. Grad-CAM++ is a class-discriminative localization technique that generates visual explanations for any CNN-based network without changing the architecture or retraining. According to the intensity of the color in the heatmap, we can know which lines of code may be more likely to be vulnerable.

4 EXPERIMENTS

In this section, we aim to answer the following research questions:

- *RQ1: What is the detection performance of VulCNN when detecting source code vulnerability?*
- *RQ2: What is the runtime overhead of VulCNN when detecting source code vulnerability?*
- *RQ3: Can VulCNN achieve large-scale vulnerability scanning?*

4.1 Experiment Settings

We first collect a dataset from *Software Assurance Reference Dataset* (SARD) [14] which is a project maintained by *National Institute of Standards and Technology* (NIST) [9]. SARD contains a large number of production, synthetic, and academic security flaws or vulnerabilities (*i.e.,* bad functions) and many good functions. In our paper, we focus on detecting vulnerability in C/C++, therefore, we only select functions written in C/C++ in SARD. Data obtained from SARD consists of 12,303 vulnerable functions and 21,057 non-vulnerable functions. Moreover, since the synthetic programs in SARD may not be realistic, we collect another dataset from real-world software. For real-world vulnerabilities, we consider *National Vulnerability Database* (NVD) [10] as our collection source. We finally obtain 1,384 vulnerable functions that belong to different open-source software written in C/C++. For real-world non-vulnerable functions, we randomly select a part of the dataset in [42] which contains non-vulnerable functions from several open-source projects. Our final dataset consists of 13,687 vulnerable functions and 26,970 non-vulnerable functions.

Four phases (*i.e.,* graph extraction, sentence embedding, image generation, and classification) in *VulCNN* are implemented by using Joern [53], sent2vec [45], networkx [15], and pytorch [16], respectively, We run all experiments on a server with 16 cores of CPU and a GTX 1080Ti GPU. For dataset, we first randomly divide it into ten subsets, then the seven subsets are used to train a CNN model, the other two subsets are used to validate, and the final subset is used to test. The metrics used to measure the effectiveness of *VulCNN* are the same as others [39, 41, 56]. For example, *true positive* (TP) reports the number of functions that are correctly classified as vulnerable and *true negative* (TN) shows the number of functions that are correctly detected as non-vulnerable.

4.2 Detection Performance

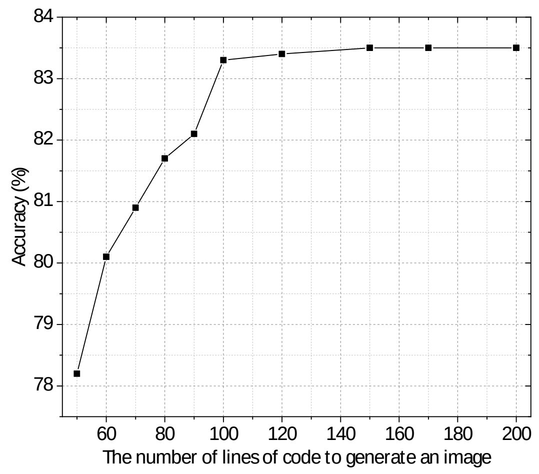


Figure 8: The detection performance of *VulCNN* with different thresholds to construct an image

We first present the detection performance of *VulCNN* under different thresholds (*i.e.,* the number of lines of code to produce our images). Because more than 99% of the functions in Figure 6 have less than 200 lines of code, we focus on the thresholds below 200 lines. Specifically, we select a total of 10 thresholds (*i.e.,* 50, 60, 70, 80, 90, 100, 120, 150, 170, 200 lines) to commence our evaluations. The experimental results are presented in Figure 8, through the results, we see that the threshold is positively correlated with detection accuracy, the larger the threshold, the higher the accuracy. However, when the threshold reaches 100 lines, the growth of accuracy becomes small. On the one hand, since the detection accuracy of *VulCNN* with 100 lines of code is almost the highest. On the other hand, the greater the threshold, the larger the image, and the more the memory required. Therefore, we select 100 lines of code as our final threshold to generate the input images.

We then compare *VulCNN* with several vulnerability detection tools, including one commercial static vulnerability detection tool (*i.e., Checkmarx* [5]), two open-source static analysis tools (*i.e., FlawFinder* [6] and *RATS* [12]), and five deep learning-based vulnerability detection approaches (*i.e., TokenCNN*[3] [47], *VulDeePecker* [41], *SySeVR* [40], *VulDeeLocator* [38], and *Devign* [56]).

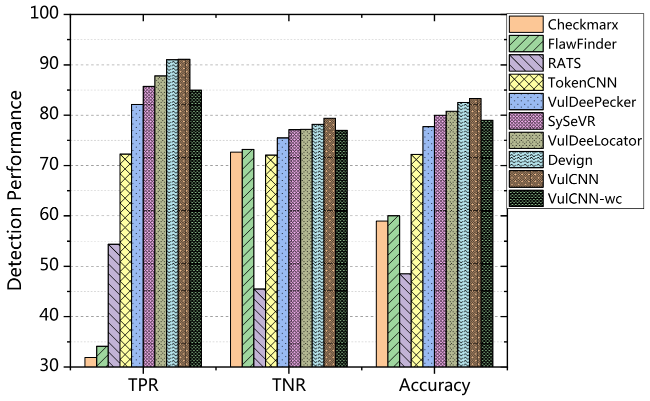


Figure 9: *True Positive Rate* (TPR), *True Negative Rate* (TNR), and Accuracy of *Checkmarx, FlawFinder, RATS, TokenCNN, VulDeePecker, SySeVR, VulDeeLocator, Devign,* and *VulCNN* on detecting vulnerability

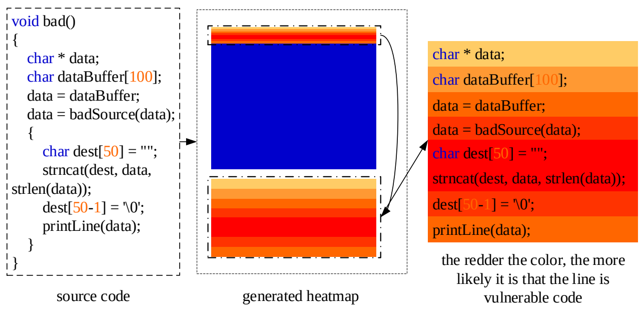
As for the commercial tool (*i.e., Checkmarx*) and two open-source static analysis tools (*i.e., FlawFinder* and *RATS*), the detection performance in Figure 9 shows that their *True Positive Rates* (TPRs) and *True Negative Rates* (TNRs) are not ideal. For example, the TPR of *Checkmarx* is only 31.9%, which means that *Checkmarx* can only detect 31.9% of vulnerabilities in our labeled dataset. Such results are reasonable because these tools depend on rules or patterns defined by human experts. In practice, there are different types of vulnerabilities, and the patterns of each type are different. In other words, experts can not define all patterns of all vulnerabilities, resulting in poor detection performance.

As for *TokenCNN*, it first conducts lexical analysis to transform the source code into a token sequence and then embeds them into fixed-length vector representations. Finally, these vectors are fed into a *Convolutional Neural Network* (CNN) model to train a vulnerability detector. Since it does not consider any program semantics, it performs worse than *VulCNN*. As for *VulDeePecker* and *SySeVR*, they both collect code gadgets by slicing programs and then transform them into corresponding vector representations. Finally, these vectors are used to train a *Bidirectional Recurrent Neural*

[3]For more convenient discussion, we call the method *TokenCNN* since it takes tokens as input and detects vulnerability by training a CNN model.

Network (BRNN) model to detect vulnerability. The difference between these two systems is that *VulDeePecker* only conducts data flow analysis to slice programs while *SySeVR* considers both control flow and data flow to obtain the program slices. Obviously, *SySeVR* performs better than *VulDeePecker* because it contains more program semantics. However, they do not consider the different contributions of different lines of code to the program semantics, but treat all lines of code in a slice as text and directly apply BRNN to train a vulnerability detector. This is the reason why they detect less vulnerability than *VulCNN*. As for *VulDeeLocator*, it first compiles a program to a LLVM bitcode file and then extracts *Intermediate Representation* (IR) slices to retain the program semantics. Finally, these IR slices are used to detect vulnerability by training a BRNN model. Since IR contains more program details than source code, *VulDeeLocator* can distinguish more vulnerability than *VulDeePecker* and *SySeVR*. However, similar to *VulDeePecker* and *SySeVR*, *VulDeeLocator* also treat the slices as text, resulting lower performance than *VulCNN*. As for *Devign*, it first applies complex program analysis to extract a graph representation that contains comprehensive program semantics and then uses a general graph neural network to detect vulnerability. The detection performance of *Devign* is almost the same as that of *VulCNN*. However, due to the complexity of the generated graph, it cannot be extended to large-scale vulnerability scanning. But *VulCNN* can, because it uses centrality analysis to convert time-consuming graphic analysis into efficient image scanning.

To check whether the use of centrality analysis contributes to *VulCNN* on detecting vulnerability or not, we construct another experiment. More specifically, after sentence embedding, we directly feed the sentence vectors into a CNN model to train a classifier without multiplying by the centrality. We call the method as *VulCNN-wc* (*i.e., VulCNN without centrality*) and show the experiment result in Figure 9. Through the figure, we can see that *VulCNN* is superior to *VulCNN-wc*, which indicates that the consideration of centrality of different lines of code in a function can improve the detection accuracy on vulnerability detection.



```
void bad()
{
    char * data;
    char dataBuffer[100];
    data = dataBuffer;
    data = badSource(data);
    {
        char dest[50] = "";
        strncat(dest, data,
strlen(data));
        dest[50-1] = '\0';
        printLine(data);
    }
}
```

```
char * data;
char dataBuffer[100];
data = dataBuffer;
data = badSource(data);
char dest[50] = "";
strncat(dest, data, strlen(data));
dest[50-1] = '\0';
printLine(data);
```

source code
generated heatmap
the redder the color, the more likely it is that the line is vulnerable code

Figure 10: The visualization of a detected buffer overflow vulnerability

Furthermore, since *VulCNN* is an image-based vulnerability detection system, we can interpret the detection results by using CNN visualization techniques. In this paper, we take *Gradient-weighted Class Activation Mapping++* (Grad-CAM++) [20] as an example to visualize our detection results. Figure 10 shows the visualization of a detected buffer overflow vulnerability. This vulnerability can be triggered when 'data' is larger than 'sizeof(dest) - strlen(dest)'. In this case, the terminal null will be written outside the buffer. Through Figure 10, we see that the color of code 'strncat(dest, data, strlen(data));' is the reddest, which means that this line of code is more likely to be vulnerable code. This result is in line with expectations.

4.3 Runtime Overhead

In this section, we perform a comprehensive evaluation on runtime overhead of *VulCNN* by using our 40,657 functions (*i.e.,* 13,687 vulnerable functions and 26,970 non-vulnerable functions). Given a new function, *VulCNN* consists of four main steps to complete the classification: *Graph Extraction*, *Sentence Embedding*, *Image Generation*, and *Classification*.

4.3.1 Graph Extraction. Given the source code of a function, the first step of *VulCNN* is to extract the PDG of it. Figure 11 presents the runtime overheads of graph extraction on our dataset, in which more than 95% functions can be obtained the graphs in three seconds. On average, it takes 1.71 seconds to construct a PDG of a function.

4.3.2 Sentence Embedding. The second step of *VulCNN* is to embed all lines of code into corresponding fixed-length vectors. As shown in Figure 11, this phase is very fast, it only requires about 0.000489 seconds to complete the sentence embedding of a PDG.

4.3.3 Image Generation. After embedding all lines of code into vector representations, the third step of *VulCNN* is to apply centrality analysis to transform the new PDG into an image. Figure 11 presents the runtime overheads of *VulCNN* in this step. More than 98% PDGs are able to be converted into images in one second, and the average runtime overhead of this phase is 0.26 seconds. Such result indicates that *VulCNN* can efficiently transform a PDG into an image.

4.3.4 Classification. The final step of *VulCNN* is to detect vulnerability using a trained CNN model. CNN classification is the fastest of all phases in *VulCNN*, it only consumes about 41 microseconds to complete the classification of an image.

In addition, we also compare the scalability of *VulCNN* with other comparative systems. We exclude the comparison of *Checkmarx*, *FlawFinder*, and *RATS* since they are not deep-learning-based methods and perform worse than the other five deep-learning-based approaches. Figure 12 shows the average runtime overheads of *TokenCNN*, *VulDeePecker*, *SySeVR*, *VulDeeLocator*, *Devign*, and *VulCNN*. As for *TokenCNN*, because it only applies simple lexical analysis to obtain the source code tokens and uses a CNN model to detect vulnerability, it is the fastest. On average, it only takes 0.25 seconds to finish the analysis of a function in our dataset. As for *VulDeePecker* and *SySeVR*, their slice generation phase needs to extract the *control flow graph* (CFG), *program dependency graph* (PDG), and *function call graph* (FCG), resulting in lower scalability than *VulCNN*. As for *VulDeeLocator*, as it is a LLVM-IR-based vulnerability detection system, the source code needs to be compiled first. The compilation phase consumes lots of time, making it difficult to detect vulnerability on a large scale. As for *Devign*, it

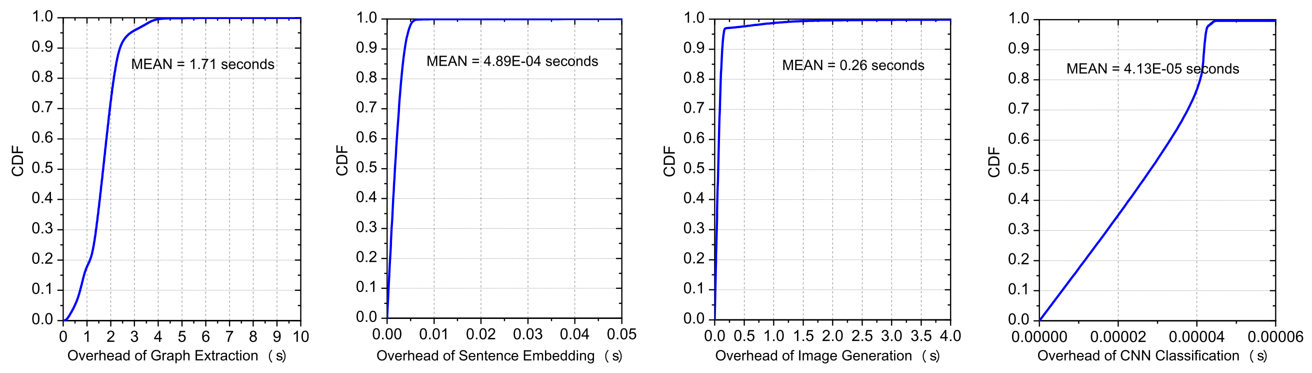


Figure 11: The *Cumulative Distribution Function* (CDF) of runtime overheads of *VulCNN* on different phases (seconds)

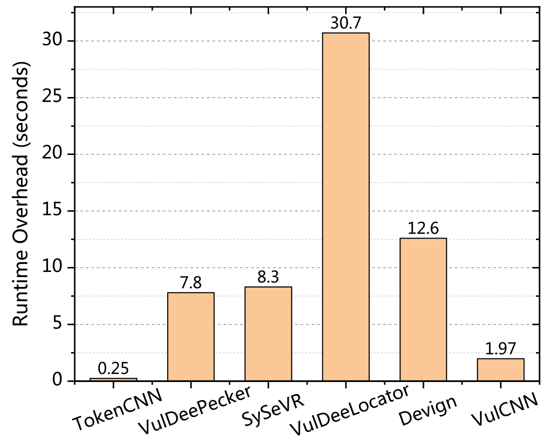


Figure 12: Comparative runtime overheads of *TokenCNN*, *VulDeePecker*, *SySeVR*, *VulDeeLocator*, *Devign*, and *VulCNN*

combines different code representations (*i.e., abstract syntactic tree* (AST), *control flow graph* (CFG), *data flow graph* (DFG), and *natural code sequence* (NSC)) to achieve accurate vulnerability analysis. Due to the heavy-weight extraction of these code representations, *Devign* consumes more runtime than *VulCNN*.

In summary, although *VulCNN* is not as scalable as *TokenCNN*, it is about four times faster than *VulDeePecker* and *SySeVR*, about 15 times faster than *VulDeeLocator*, and about six times faster than *Devign*.

4.4 Case Study

The original goal of our paper is to achieve accuracy and scalability simultaneously on detecting vulnerabilities from large-scale source code. Therefore, in this subsection, we conduct a case study to examine the ability of *VulCNN* on real-world large-scale vulnerability detection. We select three widely used open-source products as our test objects: Libav [8], Xen [17], and Seamonkey [13]. The versions of these products include both several old versions and the latest version. By this, we can report whether vulnerabilities in old versions have been "silently" patched in the latest version or not.

Table 2: Summary of our tested open-source software

OpenSource Software	#Files	#Functions	#Lines of Code
Libav-0.8.21	996	8,198	437,857
Libav-9.21	1,135	8,917	471,691
Libav-11.12	1,343	9,807	552,768
Libav-12.3	1,509	10,760	625,034
Xen-4.12.0	4,225	61,693	2,464,062
Xen-4.13.0	4,988	68,400	2,783,561
Xen-4.14.0	5,151	71,230	2,872,957
Seamonkey-2.32	11,600	153,122	6,495,189
Seamonkey-2.53.4	15,369	208,106	8,798,738
Total	46,316	600,233	25,501,857

Table 2 presents the summary of our collected products, the total number of functions that can be successfully analyzed by *Joern* [53] in these products is 600,233. In other words, *VulCNN* analyzes a total of 600,233 functions, with a total of more than 25 million lines of code.

Due to a large amount of these codes, we adopt parallel processing to analyze them. Specifically, we process ten functions at a time. The total processing runtime overheads are presented in Figure 13, it can be observed that the most time-consuming phase is to extract the PDG of functions, this phase occupies more than 87% of the total processing runtime. After generating the PDGs of these 600 thousand functions, we can transform them into corresponding images in about 253 minutes. Such high efficiency indicates that *VulCNN* has the ability to scan large-scale source code to discover new vulnerabilities.

In practice, our scanning results are also encouraging since we discover 73 vulnerabilities. Specifically, we first train a CNN model using our collected labeled 40,657 functions including 13,687 vulnerable functions and 26,970 non-vulnerable functions. Then the generated 600,233 images from more than 25 million lines of code in Table 2 are fed into the trained CNN model. After collecting all warnings reported by *VulCNN*, we then conduct manual analysis to compare them with our collected real vulnerabilities one by one. If the two are found to belong to the same pattern, it will be judged to be a real vulnerability. The analysis result shows that 73 warnings correspond to patterns of known vulnerabilities in

Table 3: 26 Vulnerabilities discovered by *VulCNN* from the latest versions of our selected products

Target product	CVE ID	Vulnerable product reported	Vulnerability release date	Vulnerable file in the target product
Libav 12.3	CVE-2011-3893	FFmpeg	11/11/2011	libavcodec/vorbis.c
	CVE-2013-0845	FFmpeg	12/07/2013	libavcodec/alsdec.c
	CVE-2013-0856	FFmpeg	12/07/2013	libavcodec/alac.c
	CVE-2015-6820	FFmpeg	09/05/2015	libavcodec/aacsbr.c
	CVE-2015-6822	FFmpeg	09/05/2015	libavcodec/sanm.c
	CVE-2015-8662	FFmpeg	12/23/2015	libavcodec/jpeg2000dwt.c
	CVE-2018-1999010	FFmpeg	07/23/2018	libavformat/mms.c
	CVE-2018-1999011	FFmpeg	07/23/2018	libavformat/asfdec.c
SeaMonkey 2.53.4	CVE-2007-5947	Firefox	11/13/2007	.../base/nsDocShell.cpp
	CVE-2008-2805	Firefox, SeaMonkey	07/07/2008	.../generic/HyperTextAccessible.cpp
	CVE-2008-2805	Firefox, SeaMonkey	07/07/2008	.../generic/Accessible.cpp
	CVE-2009-2663	Firefox	08/04/2009	.../lib/vorbis_analysis.c
	CVE-2009-3071	Firefox	09/10/2009	.../cxx/TestCrashCleanup.cpp
	CVE-2009-3071	Firefox	09/10/2009	.../cxx/TestInterruptErrorCleanup.cpp
	CVE-2010-0174	Firefox, Thunderbird, SeaMonkey	04/05/2010	.../pingsender/pingsender_win.cpp
	CVE-2014-9672	FreeType	02/08/2015	.../mac/ftmac.c
	CVE-2014-9675	FreeType	02/08/2015	.../lzw/ftlzw.c
	CVE-2014-9675	FreeType	02/08/2015	.../bzip2/ftbzip2.c
	CVE-2016-3189	Bzip2	06/30/2016	.../src/decompress.c
	CVE-2016-3189	Bzip2	06/30/2016	.../bzip2-1.0.6/bzip2recover.c
	CVE-2016-3189	Bzip2	06/30/2016	.../bzip2-1.0.6/decompress.c
	CVE-2018-5097	Firefox, Thunderbird	06/11/2018	.../xslt/txMozillaTextOutput.cpp
	CVE-2018-5181	Firefox	06/11/2018	.../widget/nsDragServiceProxy.cpp
Xen 4.14.0	CVE-2011-3346	QEMU	04/01/2014	.../scsi/scsi-disk.c
	CVE-2013-4532	QEMU	01/02/2020	.../hw/stellaris_enet.c
	CVE-2016-2841	QEMU	06/16/2016	.../hw/ne2000.c

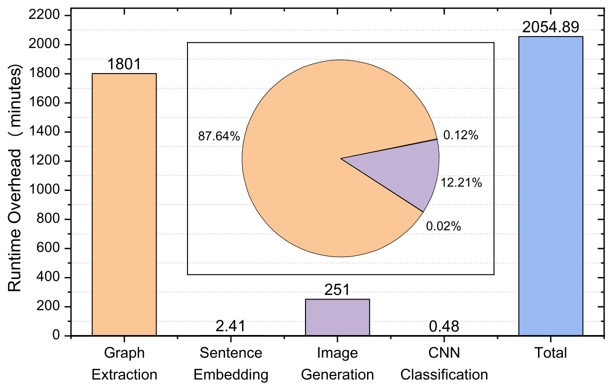


Figure 13: The total runtime overheads of *VulCNN* on processing more than 25 million lines of code

NVD. Among these detected vulnerabilities, 17 of them have been "silently" patched by vendors in the latest version of corresponding products, codes of four vulnerabilities have been deleted, and the other 52 of them still exist in the products. We have reported these vulnerabilities to their vendors and hope that they can release a patched version as soon as possible.

Due to the limited page, we show the detection results on old versions of products in our website[4] and only present the results (*i.e.,* Table 3) on three newest versions in this subsection. The details of our discovered vulnerabilities include the corresponding CVE ID in NVD, vulnerable products reported in NVD, the release data in NVD, and the vulnerable file in the target product. From six old versions of our selected products, we detect 47 vulnerabilities. From three latest versions of these products, *VulCNN* discovers 26 vulnerabilities.

5 DISCUSSION

5.1 Threats to Validity

Labeled functions in SARD are from synthetic programs. These synthetic functions may not be representative of the entire programs. To mitigate the threat, we add some real-world vulnerabilities from NVD and non-vulnerable functions from open-source products into our dataset. The selection of the maximum number of lines of a function to produce an equal-size image may cause some inaccuracies. We mitigate the threat by researching the number of lines of code for more than 40,000 functions to find a better

[4]https://github.com/CGCL-codes/VulCNN

threshold. Also, inaccuracies in detecting vulnerabilities on open-source products (*i.e.,* Libav, Xen, and Seamonkey) are inevitable since *VulCNN* may cause some false positives. The threat is mitigated by deeply comparing the pattern of detected vulnerabilities with the pattern of real-world vulnerabilities in NVD.

5.2 Discussion

5.2.1 The selection of comparative tools. We first select candidates based on whether the tool is open source, and then filter based on the type of technique used by the tool. Finally, we select one token-based tool (*i.e., TokenCNN*), one slice-based tool (*i.e., VulDeePecker*), one bitcode-based tool (*i.e., VulDeeLocator*), and one graph-based tool (*i.e., Devign*). Note that since *SySeVR* is the improved version of *VulDeePecker*, we select it as one of the comparative tools as well. Moreover, to achieve a more comprehensive comparison, we also select three traditional rule-based tools (*i.e., Checkmarx*, *FlawFinder*, and *RATS*).

5.2.2 The relationship between centrality measures and the colors of image channels. The purpose of constructing an image is to facilitate the use of image-based model (*i.e.,* CNN) for vulnerability detection while preserving the program details. Thus, our function image has three layers (the same format as regular images), but there is no rigid correspondence between three colors in regular images and three centrality measures in function images.

5.2.3 Future work. From the results in Figure 13, we see that the most time-consuming phase of *VulCNN* is to extract the PDG of functions, this phase occupies more than 87% of the total processing runtime. In our future work, we plan to design a new static analysis tool or try other static analysis tools (*e.g., Frama-C* [7]) to achieve more efficient PDG generation. In *VulCNN*, we select degree centrality, katz centrality, and closeness centrality to commence our image transformation. In practice, other different centralities can also be adopted to retain the graph details. We plan to use different combinations of different centralities to find a suitable combination to achieve more effective image transformation in *VulCNN.* Moreover, since most vulnerability detection systems are closed source, we only compare *VulCNN* with eight tools (*i.e., Checkmarx*, *FlawFinder*, *RATS*, *TokenCNN*, *VulDeePecker*, *SySeVR*, *VulDeeLocator*, and *Devign*). We will conduct detailed comparative analysis on more systems in our future work. Although *VulCNN* can maintain better effectiveness than comparative tools, its TNR is not ideal. In other words, some of our detected vulnerabilities may be false positives. In our future work, we plan to leverage directed fuzzing [19, 21] on our detected vulnerabilities to mitigate the situation.

6 RELATED WORK

There have been proposed many different vulnerability detection methods, which can be classified into two main categories: code-similarity-based and pattern-based.

As for similarity-based methods, the similarity can be measured from different aspects, such as string-based [27, 32], tree-based [29, 46], token-based [30, 48], graph-based [37], and their hybrid-based [39]. However, they can only detect cloned vulnerabilities and can not detect new vulnerabilities [41]. To tackle the challenge, pattern-based techniques have been designed.

According to the degree of automation, pattern-based work can be divided into three subcategories. 1) Manual methods: Human experts manually generate the vulnerability patterns and use them to detect new vulnerabilities. In practice, the detection effectiveness of these tools (*e.g., Checkmarx* [5], *FlawFinder* [6], and *RATS* [12]) is poor since experts can not generate all patterns of different vulnerabilities. Results in Figure 9 also demonstrate the situation. 2) Semi-automatic methods [18, 50, 51, 55]: Human experts first extract certain features (*e.g.,* subtrees and API symbols [54], imports and function calls [44]) and then feed them to traditional machine learning models (*e.g.,* support vector machine and k-nearest neighbor) to detect vulnerability. 3) More automatic methods (*i.e.,* deep-learning-based methods): Since deep learning can automatically extract features from source code, it has been used to detect source code vulnerability [22, 24, 40, 41, 43, 47, 56, 57]. For example, *VulDeePecker* [41] first collects code gadgets by slicing programs and then transform them into corresponding vector representations. Finally, it uses these vectors to train a *Bidirectional Long Short Term Memory* (BLSTM) model to detect vulnerability. *muVulDeePecker* [57] uses the program processing method in *VulDeePecker* [41] and adds a code attention to detect multi-class vulnerability. *Devign* [56] applies a general graph neural network to detect vulnerability. It contains a novel convolutional module that can effectively extract useful features from the learned rich node representation for graph-level classification. *DeepWukong* [22] distills the program semantics into a program dependency graph and splits it into several subgraphs according to the program points of interest. Then these subgraphs are fed into a graph neural network to train a vulnerability detector.

7 CONCLUSION

In this paper, we propose a novel idea that can efficiently transform the source code of a function into an image while preserving the program semantics. By this, we design a scalable graph-based vulnerability detection system (*i.e., VulCNN*). The evaluation results on a dataset of 13,687 vulnerable functions and 26,970 non-vulnerable functions report that *VulCNN* is superior to eight state-of-the-art vulnerability detectors (*i.e., Checkmarx* [5], *FlawFinder* [6], *RATS* [12], *TokenCNN* [47], *VulDeePecker* [41], *SySeVR* [40], *VulDeeLocator* [38], and *Devign* [56]). To validate the ability of *VulCNN* on large-scale vulnerability scanning, we conduct a case study on more than 25 million lines of code. Through the scanning results, we discover 73 vulnerabilities that are not reported in NVD. We have reported them to their vendors and hope that they can be patched as soon as possible.

ACKNOWLEDGEMENTS

We would thank the anonymous reviewers for their insightful comments to improve the quality of the paper. This work is supported by the Key Program of National Science Foundation of China under Grant No. U1936211 and partially supported by NSF grant CCF-2146443. We also acknowledge support for research from Siemens and Alipay.

REFERENCES

[1] 2020. 5 key takeaways from the 2020 Open Source Security and Risk Analysis report. https://securityboulevard.com/2020/05/5-key-takeaways-from-the-2020-open-source-security-and-risk-analysis-report.

[2] 2020. The Exactis Breach: 5 Things You Need to Know. https://blog.infoarmor.com/individuals-and-families/the-exactis-breach-5-things-you-need-to-know.

[3] 2020. WannaCry ransomware attack. https://en.wikipedia.org/wiki/WannaCry_ransomware_attack.

[4] 2021. Adjacency Matrix. https://en.wikipedia.org/wiki/Adjacency_matrix/.

[5] 2021. Checkmarx. https://www.checkmarx.com/.

[6] 2021. FlawFinder. http://www.dwheeler.com/flawfinde/r.

[7] 2021. Frama-C. http://frama-c.com/.

[8] 2021. Libav. https://libav.org/.

[9] 2021. National Institute of Standards and Technology. https://www.nist.gov/.

[10] 2021. National Vulnerability Database. https://nvd.nist.gov.

[11] 2021. Open-source code analysis platform for C/C++ based on code property graphs. https://joern.io/.

[12] 2021. Rough Audit Tool for Security. https://code.google.com/archive/p/rough-auditing-tool-for-security/.

[13] 2021. Seamonkey. https://www.seamonkey-project.org/.

[14] 2021. Software Assurance Reference Dataset. https://samate.nist.gov/SRD/index.php.

[15] 2021. Software for complex networks (Networkx). http://networkx.github.io.

[16] 2021. Tensors and Dynamic neural networks in Python with strong GPU acceleration (PyTorch). https://pytorch.org.

[17] 2021. Xen. https://xenproject.org/xen-project-archives/.

[18] Michael Backes, Boris Köpf, and Andrey Rybalchenko. 2009. Automatic discovery and quantification of information leaks. In *Proceedings of the 2009 IEEE Symposium on Security and Privacy (S&P'09)*. 141–153.

[19] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS'17)*. 2329–2344.

[20] Aditya Chattopadhay, Anirban Sarkar, Prantik Howlader, and Vineeth N. Balasubramanian. 2018. Grad-cam++: Generalized gradient-based visual explanations for deep convolutional networks. In *Proceedings of the 2018 IEEE Winter Conference on Applications of Computer Vision (WACV'18)*. 839–847.

[21] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. 2018. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS'18)*. 2095–2108.

[22] 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* 30, 3 (2021), 1–33.

[23] George E. Dahl, Tara N. Sainath, and Geoffrey E. Hinton. 2013. Improving deep neural networks for LVCSR using rectified linear units and dropout. In *Proceedings of the 2013 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP'13)*. 8609–8613.

[24] 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 *Proceedings of the 2019 International Joint Conference on Artificial Intelligence (IJCAI'19)*. 4665–4671.

[25] Linton C. Freeman. 1978. Centrality in social networks conceptual clarification. *Social Networks* 1, 3 (1978), 215–239.

[26] Roger Guimera, Stefano Mossa, Adrian Turtschi, and Luis A. Nunes Amaral. 2005. The worldwide air transportation network: Anomalous centrality, community structure, and cities' global roles. *Proceedings of the National Academy of Sciences* 102, 22 (2005), 7794–7799.

[27] Jiyong Jang, Abeer Agrawal, and David Brumley. 2012. ReDeBug: Finding unpatched code clones in entire OS distributions. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy (S&P'12)*. 48–62.

[28] Hawoong Jeong, Sean P. Mason, Albert L. Barabási, and Zoltan N. Oltvai. 2001. Lethality and centrality in protein networks. *Nature* 411, 6833 (2001), 41–42.

[29] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*. 96–105.

[30] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering* 28, 7 (2002), 654–670.

[31] Leo Katz. 1953. A new status index derived from sociometric analysis. *Psychometrika* 18, 1 (1953), 39–43.

[32] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. 2017. VUDDY: A scalable approach for vulnerable code clone discovery. In *Proceedings of the 2017 IEEE Symposium on Security and Privacy (S&P'17)*. 595–614.

[33] Diederik P. Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[34] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 2012 Advances in Neural Information Processing Systems (NIPS'12)*. 1097–1105.

[35] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *Nature* 521, 7553 (2015), 436–444.

[36] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *IEEE* 86, 11 (1998), 2278–2324.

[37] Jingyue Li and Michael D. Ernst. 2012. CBCD: Cloned buggy code detector. In *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*. 310–320.

[38] 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), 1–17.

[39] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Hanchao Qi, and Jie Hu. 2016. VulPecker: An automated vulnerability detection system based on code similarity analysis. In *Proceedings of the 32nd Annual Conference on Computer Security Applications (ACSAC'16)*. 201–213.

[40] 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), 1–15.

[41] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, and Yuyi Zhong. 2018. VulDeePecker: A deep learning-based system for vulnerability detection. In *Proceedings of the 2018 Network and Distributed System Security Symposium (NDSS'18)*. 1–15.

[42] Guanjun Lin, Wei Xiao, Jun Zhang, and Yang Xiang. 2019. Deep learning-based vulnerable function detection: A benchmark. In *Proceedings of the 2019 International Conference on Information and Communications Security (ICICS'19)*. 219–232.

[43] 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 (CCS'17)*. 2539–2541.

[44] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. 2007. Predicting vulnerable software components. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS'07)*. 529–540.

[45] Matteo Pagliardini, Prakhar Gupta, and Martin Jaggi. 2017. Unsupervised learning of sentence embeddings using compositional n-gram features. *arXiv preprint arXiv:1703.02507* (2017).

[46] Nam H. Pham, Tung Thanh Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. 2010. Detection of recurring software vulnerabilities. In *Proceedings of the 2010 International Conference on Automated Software Engineering (ASE'10)*. 447–456.

[47] 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 2018 IEEE International Conference on Machine Learning and Applications (ICMLA'18)*. 757–762.

[48] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. 2016. SourcererCC: Scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering (ICSE'16)*. 1157–1168.

[49] Ramprasaath R. Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. 2017. Grad-cam: Visual explanations from deep networks via gradient-based localization. In *Proceedings of the 2017 IEEE International Conference on Computer Vision (ICCV'17)*. 618–626.

[50] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David A. Wagner. 2001. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 2001 USENIX Security Symposium (USENIX Security'01)*. 201–220.

[51] Lwin Khin Shar, Lionel C. Briand, and Hee Beng Kuan Tan. 2014. Web application vulnerability prediction using hybrid program analysis and machine learning. *IEEE Transactions on Dependable and Secure Computing* 12, 6 (2014), 688–707.

[52] Yueming Wu, Xiaodi Li, Deqing Zou, Wei Yang, Xin Zhang, and Hai Jin. 2019. MalScan: Fast market-wide mobile malware scanning by social-network centrality analysis. In *Proceedings of the 34th International Conference on Automated Software Engineering (ASE'19)*. 139–150.

[53] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and discovering vulnerabilities with code property graphs. In *Proceddings of the 2014 IEEE Symposium on Security and Privacy (S&P'14)*. 590–604.

[54] Fabian Yamaguchi, Markus Lottmann, and Konrad Rieck. 2012. Generalized vulnerability extrapolation using abstract syntax trees. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC'12)*. 359–368.

[55] Fabian Yamaguchi, Alwin Maier, Hugo Gascon, and Konrad Rieck. 2015. Automatic inference of search patterns for taint-style vulnerabilities. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (S&P'15)*. 797–812.

[56] 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 *Proceedings of the 2019 Advances in Neural Information Processing Systems (NIPS'19)*. 10197–10207.

[57] Deqing Zou, Sujuan Wang, Shouhuai Xu, Zhen Li, and Hai Jin. 2019. μVulDeePecker: A deep learning-based system for multiclass vulnerability detection. *IEEE Transactions on Dependable and Secure Computing* 18, 5 (2019), 1–13.