Graph-based Vulnerability Detection via Extracting Features from Sliced Code

Peng Wu, Liangze Yin, Xiang Du, Liyuan Jia and Wei Dong
Key Laboratory Of Software Engineering for Complex Systems
College of Computer Science, National University of Defense Technology, Changsha, China
{wupeng15, yinliangze}@nudt.edu.cn, {duxiang19, jialiyuan13}@163.com wdong@nudt.edu.cn

Abstract—With the development of open source software and open source community, there are more available codes on the Internet. And the open vulnerability information can be found on the Internet. In fact, using known vulnerabilities to calculate the similarity with the source code has been demonstrated a useful method to detect vulnerabilities. But the vulnerabilities often have many irrelevant codes, which may cause false positives and reduce the accuracy of vulnerability detection. Besides, the program code may have been patched. This also leads to false positives. We use code property graphs to extract source code and calculate the similarity between the vulnerable code and the source code to judge whether the software has vulnerabilities. By using the patched code, we can reduce the false positive. We use our approach on LibTIFF and Linux kernel. The experimental results show that the approach can effectively find vulnerabilities and reduce the false positive.

Index Terms—vulnerability, patch, program slicing, code property graphs, feature extraction, similarity

I. INTRODUCTION

In recent years, when people implement a new software, they often use open source software or get code from open source community, of course this is a useful way to help them, but the quality of these codes is various. Therefore, the safety of this software will face big challenges. So how to detect vulnerabilities from the software which is implemented by open source code is an important problem. It is worth noting that the open source code often has the corresponding vulnerability information. These information maybe helps to detect vulnerabilities.

The main approaches to find vulnerabilities are static analysis and dynamic analysis. The static analysis is to analyze the code without executing the program, it is an efficient approach, but the false negative rate and false positive rate are high. The dynamic analysis is to analyze the status of the executing program, such as register contents, function execution results and memory usage. The typical dynamic approaches for example, fuzz testing and symbolic execution are unable to analyze the behavior of the entire program. Both of them don’t use open source information and require much professional knowledge.

Recent years, in order to detect software using open source code, researchers have proposed using similarity to detect vulnerabilities. That is, starting from the code segment that contains the vulnerability, and detect whether the test code is similar to the known vulnerabilities. In fact, the open source software has much vulnerability information on the Internet, such as vulnerabilities and patches in various versions. Yamaguchi et al. proposed vulnerability extrapolation [2], they think that code with a similar structure to the vulnerability is likely to be flawed. They extract the API information from the code and calculate the similarity between the test code and vulnerable code, then according to the similarity of the result help manual code review. But this method thinks about the whole function not the vulnerable code, which may cause some noise interference, and the API information can’t represent the code features. Besides, the repaired code may be similar to the vulnerable code this leads to false positive. Actually, Yamaguchi et al. also proposed code property graphs [3] that combines the Abstract Syntax Trees (AST), Control Flow Graph (CFG), and Program Dependency Graph (PDG), users can design query statements to get the information of the codes. But they don’t consider about the false positive which may lead more manual work, and the real vulnerabilities will be ignored.

After analyzing the insufficient of this similarity detection, we propose to use the code property graphs and program slicing to reduce the disturbance from the noise. We use the patch fragments to reduce the false positive. By calculating the code similarity twice, we can detect possible vulnerabilities in the source code. Generally, the code feature extraction is very important, but current methods don’t consider the key to vulnerable code instead they consider the whole functions, so that our approach extracts the key to vulnerable code by code property graphs. Besides, the patched code and the vulnerable code may be very similar. So, the patch information will play a role in reducing false positives.

In this paper, we select the open source software LibTIFF and Linux kernel for experiment. We use code property graphs extract the features of the code from the subtree of code abstract syntax tree and API (Application Programming Interface) information. Then, we expand the classical Bag-of-words model and TF-IDF model for embedding the code features to the vector space. Next, we select the vulnerability information on the CVE (Common Vulnerabilities and Exposures) website [4] and slice the vulnerable code and patches. We calculate the similarity of the source code and vulnerable code, source code and patches separately. Finally, we compare the two results to get the possible vulnerabilities. Experimental result shows that our method is useful to find vulnerabilities and reduce the false
positive. It is important to note that in this article we analyze in terms of function granularity. In summary, we make the following contributions:

- We use subtree of abstract syntax tree and API information to represent the features of code.
- We design a program slicing algorithm to remove the noise from irrelevant code.
- We expand the Bag-of-words model and vectorize the features of code with TF-IDF model.
- We use patches to reduce the false positives.
- We experiment on LibTIFF and Linux kernel to prove the effectiveness of our method.

The structure of this paper as follows: we introduce the code property graphs, the related conception about code representation in Section II. The motivation in Section III. We detailly introduce our methods in Section IV including how to use extract the features of code, embed the feature into the vector space, and use the vulnerability information. The experiment and its evaluation in Section V. The related work and conclusion in Section VI and VII.

II. PRELIMINARIES

In the field of program analysis, there are various of methods to represent the program properties and the main methods are Abstract Syntax Tree (AST), Control Flow Graph (CFG), and Program Dependency Graph (PDG). AST is a tree representation of the abstract syntax structure of the source code, the inner node represents the operator such as addition and assignment, the leaf node represents the operand such as variable and identifiers. AST does not represent every detail of the code. For example, the nested brackets are ignored and will not be presented in the form of nodes. And the if-condition-then structure can be represented by a node with two branches. AST can only be used in simple analysis due to inadequate information.

The control flow graph (CFG) is an abstract representation of a program, an abstract data structure used in the compiler, and describes all the paths that will be traversed during the execution of a program. It represents the possible flow of execution of all basic blocks in a process in the form of graph. But CFG has no data flow information, so using the CFG does not easily know the attacked data are used in which statements.

The program dependency graph (PDG) is a directed multi-graph with markers, it represents the control dependency and data dependency. PDG is built by using two types of edges: data dependent edges between variables and control dependent edges that the predicate’s influence on the variable. PDG can accurately represent the dependency relationship between the statement with the predicate, but we can no longer get the execution order of the program from the graph.

These three classical program representations can be used to describe the vulnerable features, but none of them can represent all the vulnerable features. Therefore, if we can combine these three methods the ability to represent program can be developed a lot. That is code property graphs, which can accurately represent the feature of program. Meanwhile, the code property graphs is very suitable for querying and storage with graph database. There is a sample code and its code property graphs in Fig. 1.

```plaintext
1 int fuc (int y)
2 {
3     int n = bar (y);
4     if (n==0)
5         return 1;
6     return (n+1);
7 }
```

![Fig. 1. An example of code property graphs.](image)

We can use theNeo4j graph database [5] and Gremlin graph query language [6], which is similar to the SQL language used in relational databases. Gremlin is a functional data flow language that allows users to express complex attribute graph traversals or queries in a concise manner. Each Gremlin traversal consists of a series of steps, each step is an atomic operation on the data flow. Users can use the Gremlin language to formulate the path of the query and filter the results, so that they can choose the edge according the AST, CFG, PDG, or the combination of them to traverse, which greatly improves the flexibility and sufficiency of code information acquisition.

III. MOTIVATION

When we use the vulnerability extrapolation method to calculate the similarity between the vulnerability CVE-2018-18557 in LibTIFF 4.0.9 and a repaired version of LibTIFF 4.0.9, we find that the similarity between the vulnerable function JBIGDecode and the patched JBIGDecode still has a high degree of similarity. This situation caught our attention. The vulnerability extrapolation method does not consider the use of patches, so we try to use patches and calculate the similarity again. Patches are always accompanied with the vulnerability published. We calculate the similarity between the patch of CVE-2018-18557 and patched JBIGDecode the result shows that the similarity is higher than the first calculation. We can believe if a function has high similarity with vulnerable function it may be a vulnerability, but if the similarity between this function and patches it may be a false positive.

After further analyzing we find that the vulnerability extrapolation method has another limitation. It extracts the code feature from the entire function and use it for similarity calculation. In fact, other information in the function that is
Fig. 2. The whole workflow

not related to the vulnerabilities becomes noise that affects the similarity calculation. Thus, if we can eliminate the influence of noise the result of similarity calculation will be more useful.

In order to solve this problem, we need to improve the quality of extracted feature code. That is how to describe the features of code more accurately and represent them. The function of the code is often closely related to its function call and the context structure of statement also plays an important role. Fortunately, with the help of code property graphs, we can easily use Gremlin query language to get this information. In terms of feature mapping, the paper On the Naturalness of Software [7] said that the code is natural, so it is reasonable to use the Bag-of-words model [9] for feature mapping. When fixing the vulnerability, only a little part of the statement may be modified, so those code that have been repaired may also have a high similarity with the vulnerable function, this is an obviously false positive. Therefore, we consider patches to solve this problem. That is using the patched code as the compare objects and recalculate the similarity with the others. Then compare with the two calculation results we think the function that the similarity is higher than the first calculation result maybe a false positive. In this way, we reduce the false positive and further improve the accuracy of vulnerability detection. We will describe our method in the next section.

IV. METHOD

A. PROGRAM SLICING

Program slicing is a program analysis technique used to decompose programs. It can provide programmers with program statements that are only relevant to a given point of interest. Its principles and methods were first proposed by M. Weiser [8].

The slice is important for removing noise in the vulnerable function and reduce the false positive. We slice the known vulnerable function and the patches function. Then we use the sliced vulnerable function and the patches function to calculate the similarity with other functions. Benefit from the code property graphs we can use the information from AST, CFG, and PDG.

We define a binary group \(< n, V >\) where \(n\) represents a statement and \(V\) represents a set of variables contained in \(n\), a program \(P\). And the criteria for slicing the program \(P\) is that: we select the statement that influence the according to the code property graphs. The slicing algorithm is as follows, we select the interesting statement as \(n\) then add the data dependencies nodes and control dependencies nodes by graph traversal and repeat until no new nodes are added. It should be noted that the selection of interesting nodes \(n\) needs to be done manually. After program slicing the size of the code has been greatly reduced and cut down the interference of noise. Next, we will extract the features from the code.

B. GRAPH-BASED FEATURE EXTRACTION

When the extracted feature information can well reflect the actual features of the code, the vector obtained by subsequent mapping will play a role of representation function, and it is meaningful in code similarity calculation. Thus, we need to consider which information of the code can represent the feature of the code and how to extract them. In this paper we select Application Programming Interface (API) symbol and AST information.

The API symbol usually consists of some predefined functions, and the vulnerable code is usually associated with a specific usage pattern of API. For example, the use-after-free vulnerabilities are usually related to the use of free function. The buffer-overflow vulnerabilities are related to the use of memcpy, memset, and other memory related functions. The vulnerabilities related to the use of strings are related to string
Therefore, we consider a feature extraction method that can reflect contextual information. Fortunately, the AST has been stored in the code property graphs, so it is easy to get the AST information. Every function in the program can be represented as AST so we can select the subtree to represent the features of the function. Now, we need to consider how to make the subtree to represent as many features of the function as possible and how to vectorize the tree structure.

The selection of the root node of the subtree needs follow these two points. First, the root node cannot leave out the features of the code. Second, the choice of root node should be universal, so that this method can be used for large-scale code. In the code property graph the CFG node is a subset of the AST and describe the code execution order if we select the CFG node as the root node of the AST subtree we can cover the whole function. After determining the choice of the root node of the subtree, we need to select the information from the subtree. All the information in the subtree can represent the feature of the function greatly, but this will make the similarity is hard to calculate. As we mentioned earlier the API symbol ignores the context information, so we should add the context information around the API symbol from the subtree. Such as the parameter information, condition information, variable information, and other information. In this way the subtree both represent the feature of the code and consider the code structure. A simple summary is that we use the subtree of the function’s AST to represent its features.

Next, we represent the subtree as a string for vectorization and design a corresponding algorithm2. For a given function F, the algorithm first gets all control flow nodes from F, we set these nodes are the root node of subtree. Then the algorithm uses ast2features get the features from the subtree and convert features to string. Function ast2features is a recursive processing. For example, using the ast2features for the line6 in the sample function f1 can be presented as a string "AssignmentExpr(CallExpr(func3)(CastExpr(int)))". Now we can get the features of the function and convert them to string.

Algorithm 2 Feature extraction

<table>
<thead>
<tr>
<th>Input: Given function F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output: Features of the given function S</td>
</tr>
</tbody>
</table>

\[
S ← ∅ \\
CFGNodes = F.getCFGNodes() \\
for node in CFGNodes do \\
    ASTString ← ast2string(node) \\
    S.append(ASTString) \\
end for \\

ApiList = F.getApi() \\
for api in ApiList do \\
pString ← ast2string(api) \\
    S.append(pString) \\
end for \\
\]

C. FEATURE EMBEDDING AND SIMILARITY CALCULATION

In order to calculate the similarity, we need to map the code features to vector space. Feature mapping can use function \( \phi : X \rightarrow \mathbb{R}^{|n|} \) to represent, and the function allocates a vector for each object in the space X. For each \( x \in X \), \( \phi(x) = (\phi_1(x), ..., \phi_n(n)) \) called feature vector, after that the similarity between vectors can be calculated. We use the classic Bag-of-words model for feature mapping. The Bag-of-words model is a classic mapping technique that was first used
In the field of information retrieval. It maps text documents into a vector space that composed of the words contained in the document. And the similarity of the two documents is evaluated by the inner product between the vectors.

In Bag-of-words model we use the set of documents as the input space $X$ and assume each document $x \in X$ can be represented as a word sequence of language $L$. $L$ is the set consisting of all the words in the document set. In this way, in order to map the document to the corresponding vector, each word $w \in L$ is associated with the dimension in the vector space, and the number of occurrences of the word is recorded as the corresponding value and stored. It can be represented by a mapping function:

$$\phi : X \rightarrow \mathbb{R}^{|L|}$$

The formula for calculating the value of each dimension in the vector is:

$$\phi_w(x) := w_x \ast v_w$$

(2)

The $w_x$ represents whether the word $w$ appears in the document $x$, $v_w$ represents the weight of words in the document $x$. In this way the document converts to a vector representation. To make the model more suitable for code feature mapping we adjust it a little. We select the entire functions that we want to test as the input space $X$. And for each function $x \in X$ using our feature extraction method converts them to a feature vector. The contained elements are extracted from subtree features. Besides we use the term frequency–inverse document frequency (TF-IDF) model reduce the weight of subtree features. Besides we use the term frequency–inverse document frequency (TF-IDF) model to reduce the weight of subtree features. Besides we use the term frequency–inverse document frequency (TF-IDF) model to reduce the weight of subtree features. Besides we use the term frequency–inverse document frequency (TF-IDF) model to reduce the weight of subtree features.

The process of our method as follows. First, we can easily get the source code and known vulnerability information from the Internet. Second, we use the open source tool joern to get the code property graph. Third, we use the Gremlin graph query language to get the code features. Next, we get the sliced patched code by program slicing. Then, we use the python module "gensim" for feature embedding and similarity calculation. We calculate the similarity between the known vulnerability with source code as the first result. If there are some functions have a high similarity in the first result (generally we think that the similarity above 0.5 is a high similarity), we will use the sliced patch to calculate the similarity again. Finally, we compare the two results to reduce the false positive.

A. Experimental evaluation without slicing

We select the known vulnerability from LibTIFF4.0.9 and compare its similarity with other functions to verify the feasibility of our method. We select the Linux kernel to evaluate our method in a large-scale of code.

The CVE-2018-8905 is a heap-based buffer overflow vulnerability in LibTIFF4.0.9, and located in the LZWDecodeCompat function in tif_lzw.c file. We can find that the original code exists a potential index-out-of-bounds write. By adjusting the structure and the loop conditions of the code, this vulnerability is fixed.

We extract its features from code property graphs and calculate the similarity between LZWDecodeCompat and other functions. The TABLE I shows the sort of the similarity with CVE-2018-8905.

The CVE-2018-8905 is a heap-based buffer overflow vulnerability in LibTIFF4.0.9, and located in the LZWDecodeCompat function in tif_lzw.c file. We can find that the original code exists a potential index-out-of-bounds write. By adjusting the structure and the loop conditions of the code, this vulnerability is fixed.

We extract its features from code property graphs and calculate the similarity between LZWDecodeCompat and other functions. The TABLE I shows the sort of the similarity with CVE-2018-8905.

The CVE-2018-8905 is a heap-based buffer overflow vulnerability in LibTIFF4.0.9, and located in the LZWDecodeCompat function in tif_lzw.c file. We can find that the original code exists a potential index-out-of-bounds write. By adjusting the structure and the loop conditions of the code, this vulnerability is fixed.

We extract its features from code property graphs and calculate the similarity between LZWDecodeCompat and other functions. The TABLE I shows the sort of the similarity with CVE-2018-8905.
But the LibTIFF is a small project. Now, we consider about a large-scale code Linux kernel. This time we select two vulnerabilities CVE-2017-7541 and CVE-2013-4588.

The CVE-2017-7541 is a buffer overflow vulnerability, located in brcmf_cfg80211_mgmnt_tx function in cfg80211.c. We can see that the patch adds lengths check for variable "len". And the similarity ranking in the TABLE II. The function with the similarity of 0.89 catches our attention. In fact, this is a patched function of CVE-2017-7541. Because only 4 lines of code were added during the repair, the similarity between the patched one and the CVE-2017-7541 is still high that is a false positive.

The CVE-2017-7541

The CVE-2017-7541 after slicing

The CVE-2013-4588 is a heap-based buffer overflow vulnerability, located in function do_ip_vs_get_ctl and function do_ip_vs_set_ctl in ip_vs_ctl.c. The similarity ranking in the TABLE III. The highest similarity is only 0.46, but the function with the same problem has low similarity. We can find that the reasons for the vulnerability are related to the copy_from_user function, but these high similarity functions do not call copy_from_user. This situation is due to the interference from unrelated code.

In summary, the vulnerability similarity detection method without slicing has a certain effectiveness. But it does not perform well in large and complex software projects. The overall similarity is not high and there are a lot of false positives. The main reasons for this result are the following two: 1) The appearance of the vulnerability is only related to some closely related statements in the function. The proportion of the code that not related to vulnerabilities is always high. This will lead to false positives. 2) The code to be tested has many patched functions and the repair process often only makes a few modifications.

B. Experimental evaluation with program slicing

In this section we use the method of program slicing to solve these problems. We still select the CVE-2017-7541. Because the occurrence of this vulnerability is related to the memcpy function, we select the interesting statement where the memcpy function is located. The experiment results in TABLE IV. After source code review, we find a vulnerability function brcmf_cfg80211_start_ap, and its CVE ID is CVE-2016-8658. Actually, the causes of the two vulnerabilities are similar, they do not consider the length of variable when using memcpy function.

The CVE-2013-4588

After slicing the vulnerability function, the vulnerability detection method becomes more effective. Due to the reduce of interference from irrelevant code. And analyzing the similarity ranking can reduce the work of code auditors.
C. Experimental evaluation with patches

For CVE-2017-7541 we find that whether we slice the known vulnerability function we cannot eliminate false positive for the patched function. Different from before we select the patch instead of the original vulnerability function to calculate the similarity, and the result in TABLE V. We can find that the similarity is higher than before, so we believe that this is a false positive.

<table>
<thead>
<tr>
<th>Index</th>
<th>Similarity</th>
<th>Functionname</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0</td>
<td>brcmf_cfg80211_mgmt_tx_patch</td>
</tr>
<tr>
<td>2</td>
<td>0.94</td>
<td>brcmf_cfg80211_mgmt_tx</td>
</tr>
<tr>
<td>3</td>
<td>0.20</td>
<td>write_file_regidx</td>
</tr>
<tr>
<td>4</td>
<td>0.19</td>
<td>write_file_dfs</td>
</tr>
<tr>
<td>5</td>
<td>0.17</td>
<td>ath10k_sta_tud_stats_mask_write</td>
</tr>
<tr>
<td>6</td>
<td>0.15</td>
<td>write_file_debug</td>
</tr>
<tr>
<td>7</td>
<td>0.15</td>
<td>debug_level_store</td>
</tr>
<tr>
<td>8</td>
<td>0.14</td>
<td>nx_init</td>
</tr>
<tr>
<td>9</td>
<td>0.14</td>
<td>c101_init</td>
</tr>
<tr>
<td>10</td>
<td>0.14</td>
<td>xen_net_read_rate</td>
</tr>
</tbody>
</table>

The results of the experiment show that our method has a certain effect on vulnerability detection. Compared with the vulnerability extrapolation we reduce the false positives by using program slice and patch information. And a simple conclusion in TABLE VI.

<table>
<thead>
<tr>
<th>result 1</th>
<th>result 2</th>
<th>conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>low similarity</td>
<td>low similarity</td>
<td>this maybe a vulnerability</td>
</tr>
<tr>
<td>high similarity</td>
<td>high similarity</td>
<td>this maybe a false positive</td>
</tr>
</tbody>
</table>

VI. RELATED WORK

Vulnerability detection has always been the focus of research in the security field. At present, there are a lot of methods for vulnerability detection. The static analysis technology has a long history of development and have a variety of mature analysis technology for vulnerability detection. Yamaguchi et al. proposes vulnerability extrapolation, they extract the API call information in the code and look for similar code to assist manual code review. Cassez [10] analyzes C/C++ programs based on the LLVM-IR intermediate representation generated by the Clang compiler and implement a static tool Skink. In terms of software defect prediction, Doyle and Walden [11] analyze the relationship between software metrics and vulnerabilities in the 14 widely used open source web applications between 2006 and 2008. CP-Miner [12] is based on symbol features and generates symbol sequences to check duplicate sequences in other codes, but it may cause many false positives.

For static analysis tools, open source vulnerability detection tools include RATS, Cppcheck, ITS4, CBMC, Clang [13-17], and Commercial tools include Checkmarx, Fortify, Coverity [18-20]. Checkmarx proposes a query language-based solution for identifying logical vulnerabilities and does not require to compile the source code. Fortify and Coverity need to compile the source code. Different tools are usually developed using a specific static analysis technique. Thung [21] uses five static analysis tools to evaluate the performance of the tools on three open source projects and analyzes their false positives. McLean [22] compares the defect reports from some subsets of open source software. Wagner [23] compares the defect search situation of static analysis tools in several industrial projects, and proposes a way to simply combine the search results by ignoring the false positives.

In terms of graph-based code analysis and vulnerability detection, Reps [24] shows how to represent a program in a graph and convert the program analysis into a graph accessibility problem. Chan [25] generates an API graph from the API call information. In order to satisfy the user’s search needs, an API sub graph is provided based on the information provided by user. Nguyen converts the code usage patterns to graphics. It integrates pattern mining algorithms to help people write programs and find anomalies. Yamaguchi [3] merged AST, CFG and PDG and proposed the code property graph. Users can design their own query mode according to the code property graph and analyze the source code. In this paper, we use the code property graph to analyze the source code. Scholz [26] comes up with graph reachability to analyze user input dependencies. In graph, user-controlled input is represented by the root node, if the root node, it means that the node is controlled by user input. Kinloch and Munro [27] describe the concept of a combined C graph (CCG). CCG integrates multiple program views to support understanding C code. Urma [28] uses graph query language Cypher to describe the prototype source code query system built on Neo4j. Experiments show that it can store multiple source code details at the same time, and can be expanded to millions of lines of program.

VII. CONCLUSION

In this paper we analyze a kind of vulnerability detection method by calculating the similarity with the known vulnerability. By referencing its idea and considering its limitations, we proposed a vulnerability detection method by using vulnerability information and program slicing. Our method uses code property graph to store the source code information and selects the API information and the subtree of code’s AST as code features. Then by modifying the Bag-of-words model slightly we embed the code features into vector space with TF-IDF model. Meanwhile, we use the program slicing technology cut down the noise interference from irrelevant codes. Besides, we use the vulnerability function and patch function to calculate the similarity with the source code respectively. Through the comparison of two calculation results we can remove the function that have been repaired to reduce the false positives. And the experiments show that our method effective for finding vulnerabilities and reducing false positives. But our method also exists limitations the selection of interesting
nodes is manually selected in program slicing. For complex vulnerabilities that spans many functions our method may not be useful. These problems are the focus of our post-study.

**ACKNOWLEDGMENT**

This work is supported by National Key Research and Development Program of China (No. 2018YFB0204301); and the National Nature Science Foundation of China (No. 61802415, No. 61690203, No. 61532007).

**REFERENCES**


