Computers, Materials & Continua CMC, vol.63, no.2, pp.1013-1029, 2020

A Security Sensitive Function Mining Approach Based on Precondition Pattern Analysis

Zhongxu Yin[1, *], Yiran Song[2], Huiqin Chen[3] and Yan Cao[4]

Abstract: Security-sensitive functions are the basis for building a taint-style vulnerability model. Current approaches for extracting security-sensitive functions either don't analyze data flow accurately, or not conducting pattern analyzing of conditions, resulting in higher false positive rate or false negative rate, which increased manual confirmation workload. In this paper, we propose a security sensitive function mining approach based on preconditon pattern analyzing. Firstly, we propose an enhanced system dependency graph analysis algorithm for precisely extracting the conditional statements which check the function parameters and conducting statistical analysis of the conditional statements for selecting candidate security sensitive functions of the target program. Then we adopt a precondition pattern mining method based on conditional statements nomalizing and clustering. Functions with fixed precondition patterns are regarded as security-sensitive functions. The experimental results on four popular open source codebases of different scales show that the approach proposed is effective in reducing the false positive rate and false negative rate for detecting security sensitive functions.

Keywords: Code mining, security sensitive function, function preconditions, single-linkage clustering.

1 Introduction

Security-sensitive functions are related to the common causes of vulnerability types with violations of data flow specifications, such as improper access controls, command injections [Jourdan (2009)], incorrect check of function return values, etc. Untrusted external input need to be checked before reaching parameters of security-sensitive functions to ensure security. For example, in a command injection vulnerability, if input data of the program participates in constructing command without being checked, the command line data manipulated by attackers could be executed. These types of

[1] State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou, 450001, China.

[2] Henan University of Animal Husbandry & Economy, Zhengzhou, 450046, China.

[3] University of Michigan Transportation Research Institute, Michigan, 48109-2150, USA.

[4] Zhengzhou University, Zhengzhou, 450001, China.

[*] Corresponding Author: Zhongxu Yin. Email: yinzhxu@163.com.

Received: 05 December 2019; Accepted: 18 December 2019.

vulnerabilities are taint-style vulnerabilities [Yamaguchi, Golde, Arp et al. (2014)]. Security-sensitive functions are the most important part for modeling the taint-style vulnerability. Fig. 1 shows a state transition vulnerability model based on security-sensitive functions.

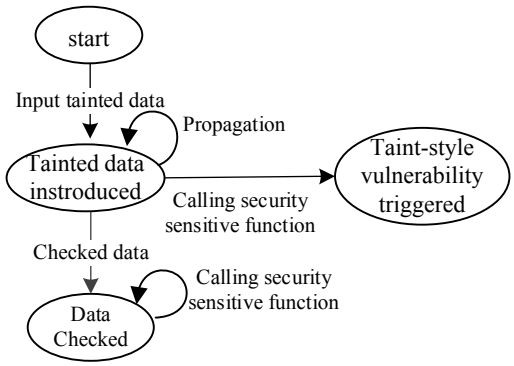

Figure 1: Taint style state transition vulnerability model

Originally, vulnerabilities detecting methods directly uses the inherent security-sensitive functions from system libraries. In 2017, Bhargava Shastry proposed a template based vulnerabilities detecting method [Shastry, Maggi, Yamaguchi et al. (2017)], which selected security sensitive functions from functions causing the vulnerabilities detected by fuzzing methods. Based on these functions, a vulnerability feature template was constructed to detecting similar vulnerabilities. Tab. 1 shows the security sensitive functions corresponding to some CVEs.

Table 1: Security sensitive functions corresponding to some vulnerabilities

Security Sensitive Functions	The target project	CVE ID
ntohs,ntohl	Open vSwitch	CVE-2017-9264
n2s	OpenSSL	CVE-2014-0160
memcpy,CopyMemory	vlc	CVE-2015-1203
atoi	pidgin	CVE-2013-6482
printf,sprintf,fprintf	xpdf	CVE-2013-4473
copy_from_user	Linux kernel	CVE-2013-6381

Research in the area of extraction of unknown security-sensitive functions with code mining [Dyer, Nguyen, Rajan et al. (2013)] has gained some achievements in past years. Typical examples include code characteristics based approaches and frequent itemset mining-based approaches. In code characteristics based approaches, security sensitive functions are extracted by mining of the pre- and post-conditions of these functions [Ramanathan, Grama and Jagannathan (2007); Nguyen, Dyer, Nguyen et al. (2015)]. AntMiner [Liang, Bian and Zhang (2016); Bian, Liang, Zhang et al. (2018)] preprocessed the source code with program slicing to reduce noise interference and filtered out security-sensitive functions through a heuristic method. Chen et al. proposed an improved approach [Chen, Yang, Liu et al. (2018, 2019)] based on implicit parameter checking to

improve the AntMiner approach, which reduced the false positive rate comparing to AntMiner. The APEX [Kang, Ray and Jana (2016)] analyzed the post-conditions of each API function called by the program, found the fallible APIs that are sensitive to error handling as security sensitive function, and identified error paths and non-error paths according to the number of branching points of the path to find error return values processing functions.

These approaches are based on analyzing of conditional statement, and the number of the check conditions of the function parameters or the return value are used to filter out the security sensitive function. As the analysis doesn't consider the common pattern of checking conditions, the methods can cause much false positives.

Some methods use cluster analysis to extract security-sensitive functions from the pattern of the clustering result. PR-Miner [Li and Zhou (2005)] uses frequent closed itemset mining techniques to mine association rules between APIs using FPclose algorithm [Grahne and Zhu (2003)], the APIs in the frequent itemset are considered as interesting APIs. The approach proposed by Chang et al. [Chang, Podgurski and Yang (2008); Chang and Podgurski (2012)] first chooses the set of APIs of interest as condidate security sensitive functions, then for every condidate API, it uses each of its call site instances to construct dependence spheres from a system dependency graph of the target program. It then performs frequent isomorphic graph minor mining from the dependence spheres. The frequent isomorphic graph minors are selected as the security-sensitive functions. Frequent graph minor mining problem for this approach is an NP-complete problem [Damaschke (1990)]. There is lack of precise analysis of the data flow in these approaches and there is a high rate of false negatives.

In this paper we propose a security sensitive function extraction approach based on code structure characteristics analysis. By conducting an improved system dependency graph analysis of the target program, the shared data dependence relationship of statements are used in the checking of protected state of parameters for selecting condidate functions. Then we cluster the protecting conditions for the condidate function to get the common pattern of checking conditions and determine a condidate function as a security sensitive functions if the common pattern exists.

The experimental results show that the proposed approach is effective in reducing the false positives and false negatives for detecting security sensitive functions.

2 Proposed method

2.1 Overview

Fig. 2 shows the overview of our approach. We first build an enhanced system dependency graph for the target source code through program analyzing. On this basis, for the parameters of each function, the verification variables in the conditional statements are extracted, and the candidate security sensitive functions are extracted according to the checking situations for parameters from the verification variables. For the candidate security-sensitive functions, the preconditions are extracted, generalized and clustered. If there is a clearly precondition checking pattern as the clustering result, the correlation function is recognized as a security-sensitive function.

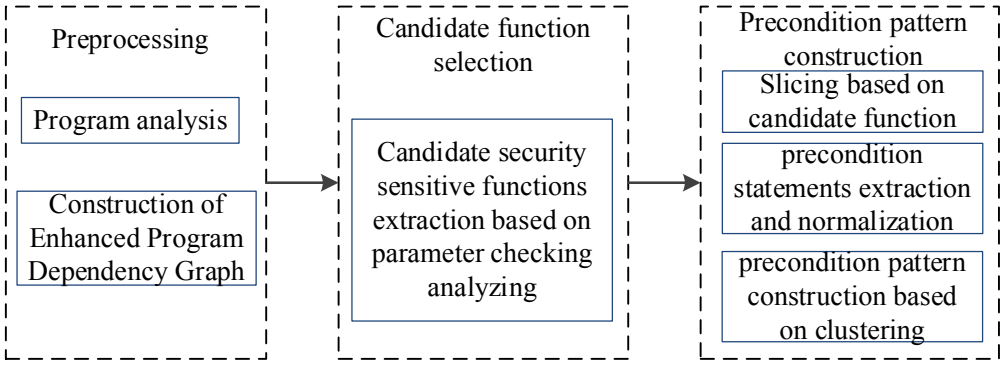

Figure 2: Overview of the proposed approach

2.2 Code processing

A precondition extraction method is used in the AntMiner method. Which performed dependency analysis on program dependency graphs (PDGs) to identify potentially error-prone functions. A set of validated variables (VVS) is calculated for each conditional statement. VVS contains all the variables checked by a conditional statement. To calculate the VVS, the data dependent subgraph (DDS) of the PDG is traversed from the conditional statement and the variables of the statements accessed during the traversal are added to the VVS. Then each calling instance of security sensitive function is examined by traversing the PDG's Control Dependent Subgraph (CDS) to see if it depends on the control condition. If such a conditional statement exists, it further check if the parameters are protected by the conditional statement. The assertion that a variable v is protected by a conditional statement is defined recursively. If the variable v belongs to the VVS of the conditional statement or another variable v' is used in the definition statement of v and v' is protected by the conditional statement, then v is considered as protected by the conditional statement. If a parameter p of a function is protected, that is, if there exists a direct check of p or a check of the variable defining p (indirect check), the protected counter of p (each parameter has a corresponding protected counter) is increased by 1. If the number of call instances that perform a check on a parameter is greater than a certain threshold, the function is recognized as a candidate of security-sensitive function.

This method mainly collects VVS from the variables of the dependency graph. It does not take into account the variables coming from the same definition. As long as any item in the set of variables from the same definition is protected, it means that all variables from the set are protected.

```
openssl-1.1.1a\crypto\sha\sha512.c
267     unsigned char *p = c->u.p;
281     size_t n = sizeof(c->u) - c->num;
283     if (len < n)
284         memcpy(p + c->num, data, len)
```

Figure 3: Code snippet 1:partial code for SHA512_Update function in OpenSSL

Fig. 3 shows the partial code associated with the memcpy call in the SHA512_Update function of OpenSSL. The first parameter of memcpy is not directly protected by the conditional statement. The parameter-related variable p and the statement c->u of line 267 have data dependencies, as found from the dependency graph, but these two variables are not part of the VVS of conditional statement in line 283 in according to the definition in AntMiner. However, the value data of the VVS variable n of the conditional statement 283 depends on c->u. In this case, both n and p share the same data depending on the same variable. In this case, the check of the relative value of the parameter is the determining factor, not the parameter itself. Therefore, as long as a parameter and a variable used for checking directly have data dependency relationship or they depend on the same variable, the corresponding conditional statement should be considered as protecting over the parameters.

The control conditions protecting the corresponding function parameters can be obtained from control dependent edges in the program dependency graph. The data dependence of the parameters to variables of control conditions can be obtained by the data dependence edges in the program dependence graph. To our best knowledge, there is no direct judging basis for the relationship of dependences on same variables for function parameters and variables of control conditions. Based on this observation, we use a representation of program that adds another shared data dependent edge to the program dependency graph, which is also used in Chang et al. [Chang and Podgurski (2012)].

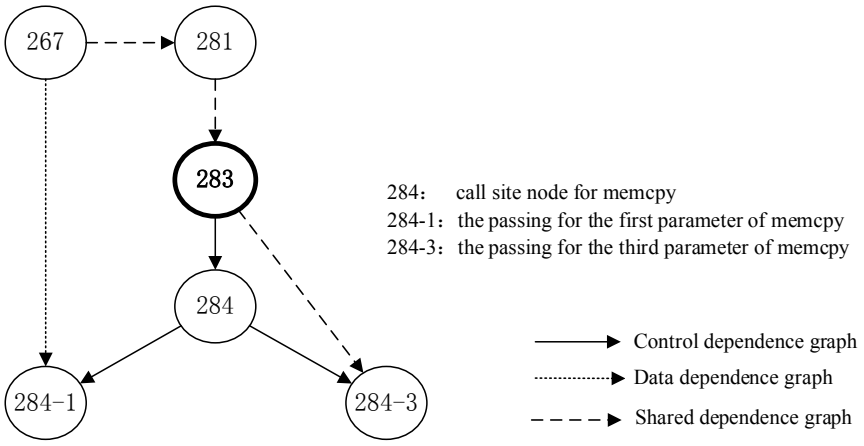

Figure 4: Enhanced system dependency graph for code snippet 1

For the two nodes a and b in the dependence graph, if both a and b use the same variable definition of c, we call a and b have shared data dependence. Program dependency graph is extended by adding a directed edge called shared data dependent edge between statements defined with the same variable in the program dependence graph. The resulting graph is called an enhanced system dependency graph (Enhanced PDG, EPDG).

In our approach we first build an EPDG for each procedure of the target program. The nodes of EPDG are statements in the program. Three types of edges are used to construct the graph: data dependence edges represent data dependence between statements, control dependence edges represent control dependence between statements, and shared data dependence edges represent shared data dependence between statements. The program is

represented by the enhanced system dependency graph (ESDG), which is constructed by the set of EPDGs of the various procedures by adding data dependent edges from actual parameter nodes of caller PDG to the formal parameter nodes of callee PDG and adding control dependence edges between the procedure call statement node of the caller PDG and the entry statement node of the callee PDG.

As an example, Fig. 4 shows an enhanced program dependence graph for the code example given in Fig. 3. If a parameter of a function and a variable of a conditional statement are connected by an SDDE edge, or a data dependent edge, it means that the parameter and the variable of the conditional statement has a data dependence or shared data dependence relationship.

2.3 Candidate function selection

The precondition of a function refers to a set of conditional statements whose related parameters must be satisfied before the function is called. The precondition is the judging conditions of the relevant parameters of a function. If there are multiple calling instances that checking the parameters of a function, we can judge that the function has preconditions, it can be used as a candidate for the security-sensitive function. We define the code characteristics representing the number of checks performed on a parameter as parameter checking feature. Since a function contains multiple call instances, the parameter checking feature of a function can be calculated by averaging the features of each objective function. The calculation formula is shown in Eq. (1), in which the $\mathrm{PCFunc}(f)$ represents the parameter checking feature of function f.

$$\mathrm{PCFunc}(f) = \frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{m} C(P_{ij}) \tag{1}$$

where m is the number of arguments of the function. n represents the number of call instances of the function. P_{ij} represents the jth argument of the ith instance of the function. $C(P_{ij})$ Represented as the number of check statements for the jth argument of the ith instance of the function.

According to this calculation method, in order to extract the parameter checking feature, it is necessary to extract the condition statements for checking parameters of the function. As the algorithm shown in Fig. 5, by traversing the enhanced data dependency graph, it is possible to determine whether a variable (function parameter or return value) is checked by the conditional statement. The input of the algorithm is the EPDG containing the SDDE edge, along with the node of the target function in the EPDG. The GetPreConditionOfFunc method starts with the the node of the target function and first traverses through the control dependent edges. Starting from the node of the corresponding parameter of the function, it collects the set of conditional statements protecting the corresponding parameters by traversing the data dependent edge and the shared data dependent edge in the graph. The protection counter corresponding to a parameter is incremented by one when encountering a checking statement of the parameter. When the protection counter exceeds the threshold, we mark the function as a candidate security sensitive function.

```
Procedure JudgeFuc(fEDPG, Ft)
Input: fEDPG: the enhanced SDG; Ft: node for call site of target function
Output: ParamProtected: BOOL array for recode the protection of
    parameters,elements of which are initialized to false
1.  ParamProtect:Array of int init to zero
2.  recursively traverse fEDPG from Ft, with CDE
3.  For each reached statement tStatement
4.     Put the statement into the conditionSet
5.  End
6.  For each call instance of Ft do
7.     For each argument argi of Ft do
8.        recursively traverse fEDPG from argi, with DDE and SDDE and
9.        For each reached statement tStatement
10.          if   tStatement ∈ conditionSet
11.               then increment ParamProtect[argi] and break the traverse
12.          Endif
13.       End
14.    End
15. End
16. For each argument argi of Ft do
17.    if  ParamProtect[argi]>MINIPROTECTED
18.        ParamProtected [argi]=TRUE
19.    Endif
20. End
```

Figure 5: Security sensitivity measurement algorithm

For example, through the traversing of the EPDG graph of the code snippet of Fig. 3. The statement in line 283 is collected as the conditional statement protecting the memcpy function in 284 line. Since there is a shared data dependence from the third parameter of memcpy to node 283, it judged that the third parameter is protected by the conditional statement. Since there is a direct data dependence edge from 267 to 284-1, and by traversing the node sequence "267->281->283" through share data dependence edges, it find that the 267 has share data dependence relationship with the conditional statement 283, so it judges that the first parameter is still protected by the conditional statement.

2.4 Precondition pattern construction

For the candidate security-sensitive functions obtained in the previous section, we extract conditional statements for checking specific parameter of the function in each calling instance.

Procedure **BackSlice**(CallGraph, A)

Input：CallGraph：call graph; A ： statement set starting for slicing

Output： S：back sliced statement set

1. Init S=∅ and Temp=A //Initialize target set and working set
2. **while** Temp ≠ ∅ do
3. Get and remove first stmt s from Temp
   //Get and remove a node from working set
4. S := S ∪ {s}
5. **foreach** edge t → s in G : //for each edge from SDG end with s
3. **if** t ∉ S then //if start node of the edge is not in slicing set
4. Temp:= Temp ∪ {t} //add it to the working set
7. **end if**
8 **end**
9. **end**

Figure 6: Backward slicing algorithm

Firstly, starting from the candidate security-sensitive function, the backward slice is performed in the enhanced system dependency graph. The algorithm for getting backward slices is listed in Fig. 6, which defines a set of sliced nodes and a set of working nodes. It first initializes the working node set to a sensitive function call statement. Through the loop, it takes out the node elements in the working node set, adds them to the slicing set, and get other related nodes with backtracking through edges in the enhanced system dependency graph. If a node that is not in the slicing set is encountered, it is added to the working node set. The loop continues until the working node set is empty. In the sliced system dependency subgraph, the definition statement of the parameter variable and the relevant conditional check statements are collected.

Then we attempt to get the check pattern over the parameter of the candidate security-sensitive function as precondtion pattern through analyzing the extracted conditional statements and judge it as a security-sensitive function if there exist at least one precondtion pattern.

The conditional statements extracted mainly include conditional expression statements and conditional statements with the keyword “if”, “for” or “while” as keywords.

When analyzing the conditional check pattern, the conditional statements are first generalized to better highlight the common features. Eliminate the influence of individual naming styles of different variables and constant. It mainly includes the following aspects:

(1) Replacing the variables relating to the corresponding parameter in the conditional statement with the serial number of the parameter, such as the name of nth parameter is replaced with the symbol “ARGn”;

(2) Replacing the variables associated with the input data in the conditional statement with the symbol “SRC”;

(3) Removing the not symbol in all statements;

(4) Uniformly processing of expressions including variables comparing to NULL. For determining whether a variable var is 0, there are three forms of judgment: "if (var!=0)", "if (0!=var)", "if (var))", and they should be converted to a consistent form;

(5) For the comparison expression, the order of the left and right subexpressions of the comparison is uniformly specified;

(6) The names of variables are replaced by the types of the variables;

(7) The relational operators are replaced by the symbol "CMP", and the numeric and constant values are replaced by "NUM";

(8) Decomposing the compound conditional expression statement. For a compound statement separated by logical symbols, the expressions are extracted, and if there is a nesting of the logical symbol, the statement is continually splitted up to get the judgment units.

Table 2: Conditional statements normalization rules

Original conditional statement	**After normalization**	**Description**
A==B,A!=B	A EQUAL B	The equivalent symbol is replaced by EQUAL, and the two sides are sorted in alphabetical order.
A	A EQUAL 0	Comparing to 0
! A	A EQUAL 0	Removing the ! symbol
A<=B,A>=B,A<B,A>B	A CMP B	The compare symbol is replaced by "CMP" symbol, the two sides are sorted in alphabetical order.
=,+,-,*,/.. (Assignment and Calculation symbols operators)	=,+,-,*,/..	Keep the original form
Parameter of sensitive function	ARGn	"n" represents the serial number of the parameter
Function return value	FUNCNAME	Replace the return value with the function name
Non-zero numeric constant	NUM	Non-zero numeric constants are replaced with "NUM" symbol
variable	type name of the variable	Replace with type name of the variable

According to these rules, the conditional statements are normalized. The processing method is shown in Tab. 2.

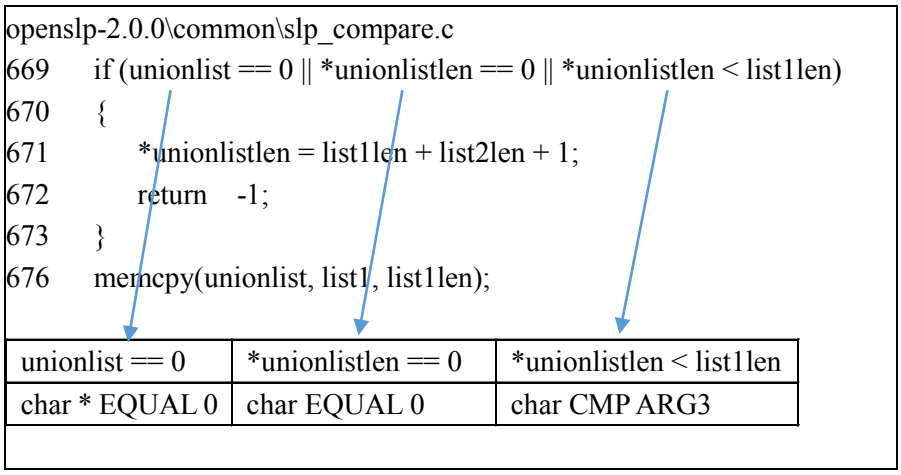

Figure 7: Conditional check statement normalization example

As shown in the program in Fig. 7, the conditional statements in line 669 is a conditional statement that checks the parameters of the "memcpy" function of line 676. During the nomalization, the conditional statement in line 669 is splitted by logical symbol to get the logical subexpressions. The results of the extraction and normalization are shown in the lower part of Fig. 7.

We scan the target program, to get all calling instances of the specified candidate security-sensitive function, represented as $\{S_1, S_2, ..., S_n\}$. For each calling instance, the enhanced system dependency graph is traversed to get all possible paths from entry node to the calling node, the path set is $P = \{P_1, P_2, ..., P_s\}$. Then we extract the set of conditional statements contained in all the paths, and collect the judgment expressions contained in all the conditional statements. The expressions are nomalized according to the above approach and aggregated to generate a conditional expression dictionary as $C = \{c_1, c_2, ..., c_t\}$.

The conditional expressions in all of the generalized conditional statements in a path are extracted and embedded into the vector space. Here we reference the natural language processing method, using the bag-of-words [Zhang, Jin and Zhou (2010)] model to vectorize the sequence. For natural language, sentence is composed of multiple words. For the conditional expression sequence, each conditional expression is referred to as "word", and the conditional expressions in a path is referred to as "sentence".

According to the inclusion of the conditional expressions in the path P of elements in C, the feature vector corresponding to the condition check mode of each path is generated. The mapping function that vectorizes P can be expressed as:

$$I(p,c)=\begin{cases}1 & \text{if conditions in p contains c}\\ 0 & \text{else}\end{cases} \quad (2)$$

For the target program, each path for each call instance of the security-sensitive function corresponds to a sequence of conditional statements which is mapped to a conditional feature vector. For the conditional feature vectors corresponding to all the conditional

expression sequences in all the calling instances, the single-linkage clustering method is used for cluster analysis. In single-linkage clustering, we need to define the distance calculation method between vectors. In this paper, the Manhattan distance is used to obtain the conditional clusters with similar conditions.

The number of elements of a cluster represents the significance level of the feature of the cluster. Smaller clusters represents pattern only supported by a few call instances. The corresponding condition expressions contained in the path cluster whose number of supporting paths exceeds the specified threshold is taken as a typical precondition pattern of the corresponding function. If there in no cluster whose number of paths exceeds the threshold, the candidate function is not recognized as a security sensitive function.

3 Results and disscussion

3.1 Implementation

We implement our approach based on Joern [Yamaguchi, Wressnegger, Gascon et al. (2013)]. Joern is an open source project for extracting and analyzing code property graphs of C/C++ code. We use it to extract data dependent edges and control dependent edges in code property graphs across procedures, and obtain system dependency graphs. We improved it by adding the shared data dependence edge of the system dependency graph to obtain an enhanced system dependency graph. On this basis, the security sensitive function recognition based on parameter checking measurement and conditional expression extraction and clustering operations are implemented.

In the experiment, we first select the relevant CWE (Common Weakness Enumeration) sample code in the static analysis benchmark analysis tool published by the NIST SAMATE for the evaluation for parameter setting of the clustering algorithm. The samples contain a number of synthetic programs, each of which has one good and bad program and covers various type of CWEs. Further more, among the samples corresponding to these types of vulnerabilities, the characteristics of the conditional checking statements are obvious and can be easily extracted for manually verifying the correctness of the recognization of security sensittive functions in our approach.

Then four open source projects, Openslp-2.0.0, LibTIFF-4.0.10, httpd-2.4.39 and OpenSSL-1.1.1 were used to extract the security-sensitive functions, and the effectiveness of the method was evaluated.

3.2 Conditional expression clustering algorithm and parameter selection

Using the program static analysis benchmark analysis tool Juliet C/C++ test suite (Version 1.2) released by the NIST SAMATE project, the parameter selection analysis for conditional expression clustering in our approach is carried out. The test set contains a variety of CWE vulnerability types, each of which contains a collection of test cases. In this paper, the positive sample data of the four types of CWE models in the test data set are selected, as displayed by Tab. 3.

Table 3: CWE data of NIST SAMATE project used for testing

CWE ID	The target project	Instances	Security sensitive function
CWE252	Unchecked Return Value	630	printLine
CWE253	Incorrect Check of Function Return Value	684	printLine
CWE78	OS_Command_Injection	8200	EXECV EXECL
CWE15	External Control of System or Configuration Setting	36	SetComputerNameA

The conditional expressions of the candidate security sensitive function related paths are extracted and normalized. After the vectorizing of the expressions based on the bag-of-words model, the results of clustering by single-linkage clustering and complete-linkage clustering algorithm are compared. When the Manhattan distance parameter for the clustering algorithm is set to a different value from 0 to 4, the respective false alarm rate and false negative rate are calculated, and the performance receiver operating characteristic curve (ROC) is obtained under two different algorithms, as shown in Fig. 8. It can be seen that, overall, the single-linkage algorithm performs better, and when the selected maximum distance is 2 or 3, the positive rate is already close to 100%. In practical applications, according to different software complexity, we select the distance for clustering with 2 or 3 for single-linkage algorithm.

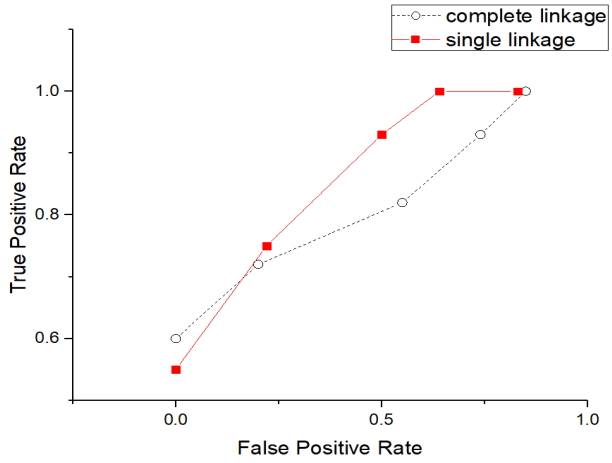

Figure 8: ROC curve for different clustering algorithms for conditional statements

When comparing the different result for the two clustering algorithm, we found that in complete linkage clustering, the checking pattern in the clustering result tend to have more conditional expressions while the clustering set is smaller than that of single-linkage clustering, although the latter tend to result in less conditional expressions in the checking pattern. That is because in single-linkage clustering, the similarity of two clusters is the similarity of their most similar members. The algorithm paid attention

solely to the area where the two clusters come closest to each other. While in complete-linkage clustering [Hubert (1974)], the similarity of two clusters is the similarity of their most dissimilar members.

As the goal of our approach is to judge precisely whether the parameters of a function has a fixed checking pattern, not to find the checking pattern itself. While finding the checking pattern is the goal of the approach in Yamaguchi et al. [Yamaguchi, Maier, Gascon et al. (2015)], which uses complete-linkage clustering to find checking pattern with as much conditional expressions as possible.

3.3 Validation of the approach

In this section, two experiments are designed to verify the validation of the proposed method: firstly we test the verification of candidate security sensitive function selecting approach, then we test the effect of the conditional expression clustering for the confirmation of candidate security sensitive functions.

3.3.1 Candidate security sensitive function extraction

For each of the four open source projects, Openslp-2.0.0, httpd-2.4.39, LibTIFF-4.0.10, and OpenSSL-1.1.1, extract the parameter checking feature of the relevant functions. Then sort the sensitive functions according to the value of the feature, and select the functions with PCFunc value exceeding a specific threshold as the candidate sensitive function. Tab. 4 shows partial of the candidate security-sensitive function and their PCFunc values. We set the threshold as 0.5 especially.

Table 4: Partial candidate security-sensitive functions extracted from four target projects

The target project	The candidate security sensitive functions	Instances	PCFunc value
Openslp	memcpy	267	1.45
	SLPUnionStringList	6	0.5
	ReadFileProperties	3	1
httpd	strcasecmp	786	0.98
	dav_lookup_uri	4	0.75
LibTIFF	_TIFFmemcpy	120	0.85
	_TIFFmemcmp	4	0.5
OpenSSL	EVP_PKEY_copy_parameters	15	0.8
	OPENSSL_strlcpy	32	0.87

We can find that the PCFunc value of memcpy in Openslp is more than 1.45. The reason is that more than one parameter is protected by conditional expressions in most calling instances.

3.3.2 Validation analysis

In order to verify the validation of the approach, from each of the four target programs, we select one target module for analyzing. As shown in Tab. 5. In total 28 security-sensitive functions are manually selected as the positive samples, while 28 ordinary functions are selected as negative samples.

Table 5: Modules of target project used for validation analysis of the proposed approach

Project name	Module	Relative path of the module
Httpd-2.4.39	moddav	httpd-2.4.39\modules\dav
openslp-2.0.0	libslp	openslp-2.0.0\libslp
LibTIFF-4.0.10	tiff2pdf	tiff-4.0.10\tools
OpenSSL-1.1.1	timestamp	openssl-1.1.1\crypto\ts

The indicators for evaluating the approaches is as shown in Tab. 6. True Positive (TP) and True Negative (TN) belong to the correct classification of samples, and False Positive (FP) and False Negative (FN) belong to the case where the samples are misclassified.

Table 6: Evaluation indicators of the proposed approach

Test results / Actual results	Positive	Negative	Total
Positive	TP	FN	TP+FN
Negative	FP	TN	FP+TN
Total	TP+FP	FN+TN	TP+TN+FP+FN

The true positive rate (TPR) is the ratio of the number of positive samples found to the total number of positive samples, also called the recall rate. The negative positive rate (FPR) is the ratio of the number of negative samples that are falsely reported as positive samples to the number of negative samples. The calculation formula is:

$$TPR = \frac{TP}{TP + FN}, \quad FPR = \frac{FP}{FP + TN} \tag{3}$$

The analysis results are as shown by Fig. 9. It can be seen that, overall, the approach proposed is superior to the AntMiner approach in terms of TPR and FPR indicators. We can see that in our approach, the testing result for LibTIFF both have lower TPR and higer FPR. In deeply analysis of the reasons we found that too much error processing functions are recognized as security sensitive functions in LibTIFF, which leads to higer FPR. We also found that several positive samples were even not selected as candidate security sensitive functions. As the parameters were security checked not by simply conditional expressions but by specified functions in LibTIFF, and in our approach we didn't considered the specific functions for checking conditions. This is closely related to the characteristics of the target project.

By comparing the extracting result of our approach and the AntMiner approach, we

analyzed the reason for the lower FPR and higher TPR. The lower FPR comes from that our approach has filtered the randomly judging operations by clustering the nomalized conditional expressions and relys on the existing of fixed checking pattern to recognize the function as a security sensitive function. The higher TPR comes from that the proposed approach used the shared data dependency relationship for extracting the VVS, and extracted the conditional expressions including variables having shared data dependency relationship with the parameter of the function under test. Which were omitted by the AntMiner approach.

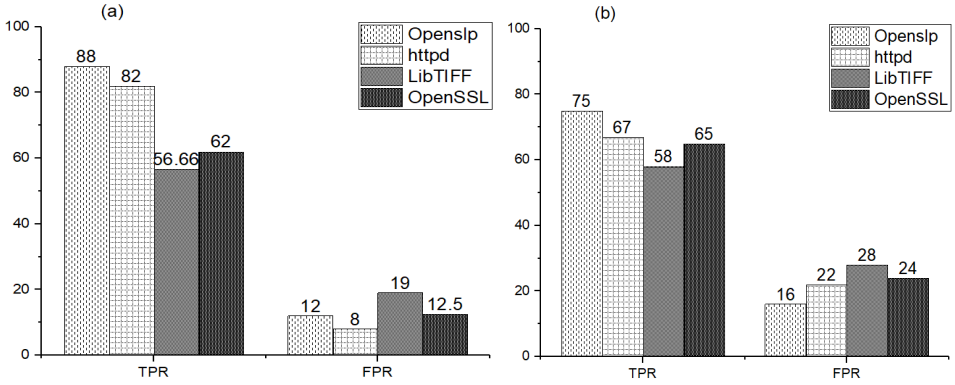

Figure 9: Comparison of the proposed approach and the AntMiner approach (a) the proposed approach (b) the AntMiner approach

4 Conclusions

This paper proposes a security sensitive function mining approach based on precondition analysis. We propose an algorithm for extracting and conducting statistical analysis of the conditional statements protecting function parameters to obtain condidate security sensitive functions. We constructed the enhanced system dependency graph containing the shared data dependent edges for extracting the protecting conditional statements to reduce the false negative rate. Then we proposed a precondition pattern mining method based on condition statements nomalizing and clustering. In the end, functions with fixed parameter checking pattern are regarded as security-sensitive functions. The experimental results on four popular open source code bases of different scales show that the method proposed is effective in reducing the false positives and false negatives for detecting security sensitive functions.

For the testing of some target projects, higer false positive rate and lower true positive rate were caused by the fact that the check conditions were hidden in a special function and the error handling functions is regarded as a security sensitive function. The future work is to improve the approach by identifying and processing the specific checking functions and the error handling functions.

Acknowledgement: This work was supported by the National Key R&D Program of China (Grant No. 2016QY07X1404), the Zhejiang Provincial Natural Science Foundation of

China (Grant No. LY19E050012), the Humanities and Social Sciences project of the Ministry of Education of China (Grant No. 19YJCZH005).

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

Bian, P.; Liang, B.; Zhang, Y.; Yang, C.; Shi, W. et al. (2018): Detecting bugs by discovering expectations and their violations. *IEEE Transactions on Software Engineering*, vol. 45, no. 10, pp. 984-1001.

Chang, R. Y.; Podgurski, A. (2012): Discovering programming rules and violations by mining interprocedural dependences. *Journal of Software: Evolution and Process*, vol. 24, no. 1, pp. 51-66.

Chang, R. Y.; Podgurski, A.; Yang, J. (2008): Discovering neglected conditions in software by mining dependence graphs. *IEEE Transactions on Software Engineering*, vol. 34, no. 5, pp. 579-596.

Chen, L.; Yang, C.; Liu, F.; Gong, D.; Ding, S. (2018): Automatic mining of security-sensitive functions from source code. *Computers, Materials & Continua*, vol. 56, no. 2, pp. 199-210.

Chen, L.; Yang, C.; Liu, F.; Gong, D.; Ding, S. (2019): A security-sensitive function mining framework for source code. *Proceedings of International Conference on Artificial Intelligence and Security*, pp. 421-432.

Damaschke, P. (1990): Induced subraph isomorphism for cographs is NP-complete. *Proceedings of International Workshop on Graph-Theoretic Concepts in Computer Science*, pp. 72-78.

Dyer, R.; Nguyen, H. A.; Rajan, H.; Nguyen, T. N. (2013): Boa: a language and infrastructure for analyzing ultra-large-scale software repositories. *Proceedings of International Conference on Software Engineering*, pp. 422-431.

Grahne, G.; Zhu, J. (2003): Efficiently using prefix-trees in mining frequent itemsets. *Proceeding of IEEE ICDM Workshop on Frequent Itemset Mining Implementations*, pp. 123-132.

Grahne, G.; Zhu, J. (2003): High performance mining of maximal frequent itemsets. *Proceedings of the 6th SIAM International Workshop on High Performance Data Mining*, pp. 135-143.

Hubert, L. (1974): Approximate evaluation techniques for the single-link and complete-link hierarchical clustering procedures. *Journal of the American Statistical Association*, vol. 69, no. 347, pp. 698-704.

Jourdan, G. V. (2009): Securing large applications against command injections. *IEEE Aerospace and Electronic Systems Magazine*, vol. 24, no. 6, pp. 15-24.

Kang, Y.; Ray, B.; Jana, S. (2016): APEX: automated inference of error specifications for c apis. *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pp. 472-482.

Li, Z.; Zhou, Y. (2005): PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. *Proceedings of the 10th European Software Engineering Conference and the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 306-315.

Liang, B.; Bian, P.; Zhang, Y. (2016): AntMiner: mining more bugs by reducing noise interference. *Proceedings of the 38th International Conference on Software Engineering*, pp. 333-344.

Nguyen, H. A.; Dyer, R.; Nguyen, T. N.; Rajan, H. (2015): Consensus-based mining of API preconditions in big code. *Proceedings of the 2015 ACM International Conference on Systems, Programming, Languages and Applications: Software for Humanity*, pp. 5-6.

Ramanathan, M. K.; Grama, A.; Jagannathan, S. (2007): Static specification inference using predicate mining. *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation,* 123-134.

Shastry, B.; Maggi, F.; Yamaguchi, F.; Konrad, R.; Seifert, J. P. (2017): Static exploration of taint-style vulnerabilities found by fuzzing. *Proceedings of the 11th USENIX Workshop on Offensive Technologies*, arXiv:1706.00206.

Yamaguchi, F.; Golde, N.; Arp, D.; Rieck, K. (2014): Modeling and discovering vulnerabilities with code property graphs. *Proceedings of IEEE Symposium on Security and Privacy*, pp. 590-604.

Yamaguchi, F.; Maier, A.; Gascon, H.; Rieck, K. (2015): Automatic inference of search patterns for taint-style vulnerabilities. *Proceedings* of *IEEE Symposium on Security and Privacy*, pp. 797-812.

Yamaguchi, F.; Wressnegger, C.; Gascon, H.; Rieck, K. (2013): Chucky: exposing missing checks in source code for vulnerability discovery. *Proceedings of ACM SIGSAC Conference on Computer & Communications security*, pp. 499-510.

Zhang, Y.; Jin, R.; Zhou, Z. H. (2010): Understanding bag-of-words model: a statistical framework. *International Journal of Machine Learning and Cybernetics*, vol. 1, no. 1-4, pp. 43-52.