Copyright © 2018 Tech Science Press CMC, vol.56, no.2, pp.199-210, 2018

Automatic Mining of Security-Sensitive Functions from Source Code

Lin Chen[1, 2] , Chunfang Yang[1, 2, *], Fenlin Liu[1, 2], Daofu Gong[1, 2] and Shichang Ding[3]

Abstract: When dealing with the large-scale program, many automatic vulnerability mining techniques encounter such problems as path explosion, state explosion, and low efficiency. Decomposition of large-scale programs based on safety-sensitive functions helps solve the above problems. And manual identification of security-sensitive functions is a tedious task, especially for the large-scale program. This study proposes a method to mine security-sensitive functions the arguments of which need to be checked before they are called. Two argument-checking identification algorithms are proposed based on the analysis of two implementations of argument checking. Based on these algorithms, security-sensitive functions are detected based on the ratio of invocation instances the arguments of which have been protected to the total number of instances. The results of experiments on three well-known open-source projects show that the proposed method can outperform competing methods in the literature.

Keywords: Code mining, vulnerabilities, static analysis, security-sensitive function, source code.

1 Introduction

With the development of the Internet, information security has attracted widespread attention. In recent years, researchers have researched several related topics, such as cryptography [Kosba, Miller, Shi et al. (2016); Terzi, Terzi and Sagiroglu (2015)], information hiding [Luo, Song, Li et al. (2016); Ma, Luo, Li et al. (2018); Zhang, Qin, Zhang et al. (2018)] and software security [Stallings and Brown (2015)]. Vulnerabilities are a major threat to software security. Severe vulnerabilities have caused global hazards in the recent past, such as the "Heartbleed" vulnerability in a cryptographic library that caused a massive leak of private information [MITRE (2014)], and the WannaCry ransomware virus [Wikipedia (2017)] that exploited vulnerability MS17-010 [Microsoft (2017)] in Windows. The number of computer vulnerabilities worldwide continues to rise. From 2000 to 2017, the number of vulnerabilities identified every year has increased from 1,000 to 14,000 [Özkan (2017)].

To detect software vulnerabilities, a variety of automated vulnerability detection

[1] Zhengzhou Science and Technology Institute, Zhengzhou, 450001, China.

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

[3] University of Göttingen, Goldschmidtstr. 7, 37077 Göttingen, Germany.

* Corresponding Author: Chunfang Yang. Email: chunfangyang@126.com.

techniques have been proposed, such as fuzzing tests [Rawat, Jain, Kumar et al. (2017); Wang, Wei, Gu et al. (2010)], model checking [Engler and Musuvathi (2004)], symbolic execution [Godefroid, Klarlund and Sen (2005); Cadar, Dunbar and Engler (2008); Godefroid, Levin and Molnar (2008)] and machine learning [Menzies, Greenwald and Frank (2007); Yamaguchi, Lindner and Rieck (2011)]. However, the structure and scale of practical software are becoming increasingly complex. When dealing with large-scale software, the above techniques encounter such problems as path explosion, state explosion, high time complexity, and low efficiency. In light of the requirements of vulnerability detection in large-scale software, the problem can be divided into subclasses or subsets. For example, Yamaguchi et al. [Yamaguchi, Maier, Gascon et al. (2015)] proposed an automatically generated detection script for a given sink function to search for taint-style vulnerabilities on a code property graph. The guided fuzzer, which uses critical operations as starting point for testing, was proposed in Ganesh et al. [Ganesh, Leek and Rinard (2009); Haller, Slowinska, Neugschwandtner et al. (2013)], and yielded good fuzzer efficiency. AntMiner [Liang, Bian, Zhang et al. (2016)] proposed a detection technique that searches for bugs in sub-repositories of the original source code of a given software.

The above automatic detection techniques improve the efficiency of detection by scale decomposition. Such segmentation at present is often based on a security-sensitive function or a type of security-sensitive function. Thus, the identification of security-sensitive functions is an important step in many automatic vulnerability detection methods. Many security vulnerabilities still depend on manual detection, which involves attending closely to high-level security-sensitive functions. Therefore, the identification of security-sensitive functions is significant for automatic detection techniques and manual detection.

Manual analysis is the most common method for the identification of security-sensitive functions these days. Of functions in the common libraries, security-sensitive ones, such as *memcpy()* and *strcpy()* in the C library, are identified by experience. However, in case of self-implementing functions of code projects (such as *BlockMove()* and SSL*_free()* in OpenSSL), the efficiency of manually identifying security-sensitive functions is low. In such problem, project documents are used to determine whether a function is security sensitive. However, document-based analysis requires the source code, and is challenging to apply when the documents are not clear or do not exist. Thus, manual identification of security-sensitive functions is a tedious task, especially for large-scale programs.

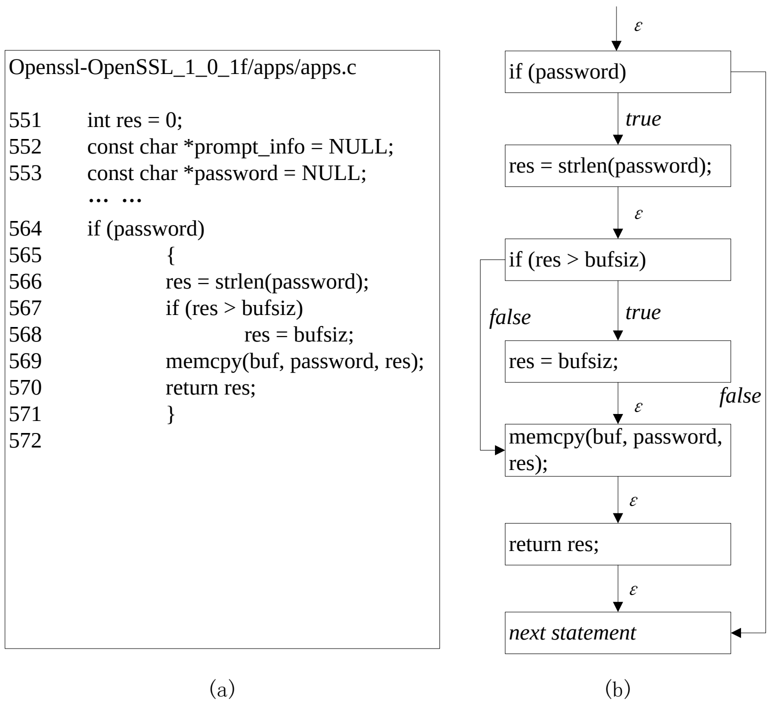
Research on automatic mining of security-sensitive functions is scarce. Ganesh et al. [Yun, Min, Si et al. (2016)] analyzed the semantic relationship among arguments when inferring correct API usage from the source code, which provides some inspiration for the method proposed here. AntMiner [Liang, Bian, Zhang et al. (2016)] mined potentially bug-prone functions by checking arguments protected by a conditional statement. AntMiner considered only one implementation of argument checking and thus omitted some security-sensitive functions. Past research has shown that security-sensitive functions can be automatically mined, but due to the differences in the forms of these functions, many types of security-sensitive functions are ignored in the mining process.

Based on the work in Liang et al. [Liang, Bian, Zhang et al. (2016); Yun, Min, Si et al. (2016)], this study proposes an improved method to automatically mine security-sensitive functions. The proposed method considers the common form of argument checking as well as constraint checks between arguments, thus revealing more security-sensitive functions than AntMiner. The results of experiments show that the proposed method outperforms that proposed in AntMiner.

2 Argument-sensitive functions

A security-sensitive function is a function with security specifications. Failure to satisfy any of the specifications to call this function may result in bugs in the program. The forms and implementations of a security specification are diverse. One type of security-sensitive functions attributes their security to the validity of their arguments. The form of security specification, in this case, is that the arguments of the function instance need to meet specific legal requirements, such as that the argument cannot be NULL or must be positive, and the implementation is that the caller of the function needs to ensure that this argument is legal.

This article calls the above type of security-sensitive function an argument-sensitive security-sensitive function, or simply an argument-sensitive function. Argument-sensitive functions such as *memcpy()*, *strcpy()* in the early C library are used in many programs. If the callers want to safely call argument-sensitive functions, they need to check the validity of the incoming arguments. There are two main implementations:



```
Openssl-OpenSSL_1_0_1f/apps/apps.c

int res = 0;
const char *prompt_info = NULL;
const char *password = NULL;
… …
if (password)
        {
        res = strlen(password);
        if (res > bufsiz)
                res = bufsiz;
        memcpy(buf, password, res);
        return res;
        }
```

Figure 1: Code snippet of OpenSSL and its CFG

(1) Explicit checking. The caller uses conditional statements to determine whether the argument satisfies the security requirements. If the condition is not satisfied, it means that the argument does not meet the security specifications, and the function is not executed.

Fig. 1 shows a code snippet of OpenSSL and its control flow graph, which illustrates two argument-checking methods. Fig. 1(a) calls two argument-sensitive functions, *strlen()* and *memcpy()*, on lines 566 and 569, respectively. The function prototype of *strlen()* is *strlen(char *s)*, and is used to obtain the length of string s. Its security specification is that the parameters cannot be a null value, as this may result in undefined and unsafe behavior under certain circumstances. The function prototype of *memcpy ()* is *memcpy (void *dest, const void *src, size_t n)*, and is used to copy data of length *n* from *src* to *dest*. The security specification of this function is more complex, and requires that the parameters *dest* and *src* not have a *null* value, and that the size of *dest* and *src* must be greater than or equal to *n*; otherwise, it may cause buffer overflow. It is clear that if the caller wants to safely call *strlen()* and *memcpy()*, it must check the validity of the arguments.

For the invocation instance of the function *strlen(), res=strlen(password)* (on line 566), statement 564 checks whether the argument is *null* in the conditional statement. If the check fails, the invocation instance is not executed. Therefore, this is an explicit check on the argument "*password*."

For the invocation instance of the function *memcpy(), memcpy(buf, password, res)* (on line 569), statement 564 is an explicit check on the argument "*password*." While conditional statement 567 also checks the third argument "*res*" it cannot control the execution of the instance, and thus this is not an explicit check.

(2) Implicit checking. This does not directly check the validity of arguments, but obtains legal values by means of assignment or calculation and passes them as arguments to satisfy the security specifications of function calls.

For the invocation instance of the function *memcpy(), memcpy(buf, password, res)*, through the control flow graph (CFG) in Fig. 1(b), two values of argument *res* under two execution paths can be obtained: *res=strlen(password)* (on line 566), and *res = bufsiz* (on line 568). In the first case, the value of *res*, is equal to the length of *password*, which is the second argument in *memcpy (buf, password, res)*. It thus satisfies the security specification whereby the value of the third argument in *memcpy()* is smaller than the size of the second argument. Statement 566 is thus an implicit check on *res* and *password*. In the second case, as the source of the value of *bufsize* cannot be determined from the code snippet, and although *bufsize* may represent the size of *buf*, it cannot be determined whether the assignment of *res* on line 568 satisfies the security specification of *memcpy()*. Therefore, it cannot be determined whether this is an implicit check on the argument *res*.

As shown in the above examples, both explicit and implicit checks can implement safety specifications for argument-sensitive functions. And explicit checks have more obvious characteristics, whereas implicit checks are more subtle and difficult to identify.

3 Mining argument-sensitive functions based on check identification

To prevent potential security problems, the validity of arguments should be checked before argument-sensitive functions are called. A mature program should have been tested repeatedly by developers and testers. Moreover, over a long-term deployment, developers continue to repair and update security issues. Therefore, following multiple iterations, most calls of argument-sensitive functions in a given program can be considered correct, and their arguments are checked for validity. That is, for a given function, the higher the percentage of invocation instances the validity of which has been verified, the higher the probability that the function is an argument-sensitive function.

Based on the above principle, this section provides a method to mine argument-sensitive functions based on the validity check proportion as shown in Fig. 2. The method consists of two parts. The first analyzes each invocation instance of the function and uses the identification algorithm to determine whether each argument has an explicit or an implicit check to obtain check information. The second part uses a decision algorithm to determine whether the given function is an argument-sensitive function based on the total number of invocation instances, the check information of each function instance and a threshold.

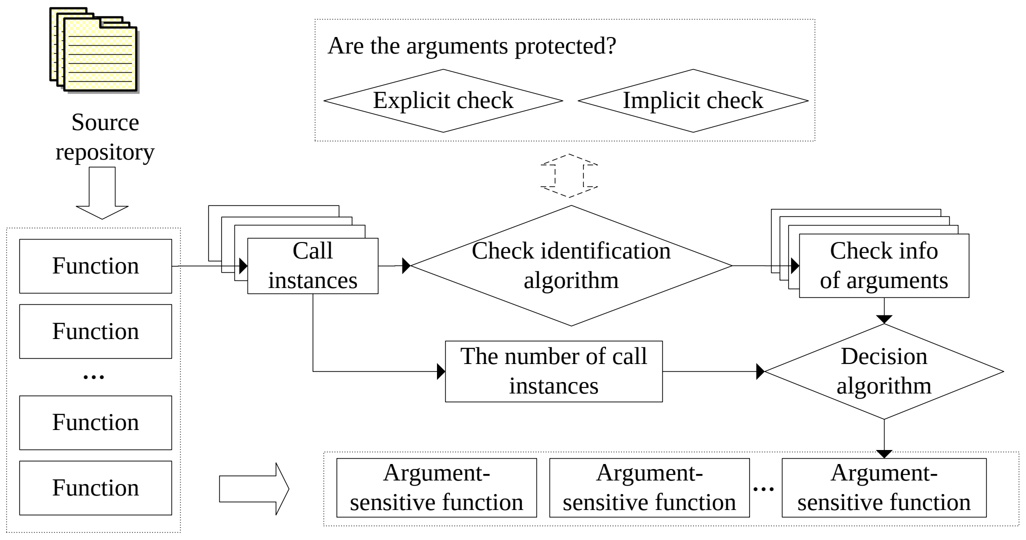

Source repository
Are the arguments protected?
Explicit check
Implicit check
Function
Function
...
Function
Function
Call instances
Check identification algorithm
Check info of arguments
The number of call instances
Decision algorithm
Argument-sensitive function
Argument-sensitive function
...
Argument-sensitive function

Figure 2: Mining argument-sensitive functions based on check identification

3.1 Explicit check identification algorithm

The validity of the arguments is checked by a conditional statement during the explicit check. If it passes this check, the function instance is subsequently called; otherwise, it is not called. Based on this feature, AntMiner identified whether explicit checks were performed prior to the function call. To consider the overall efficiency of the identification of argument-sensitive functions, this section describes improvements to the explicit check of AntMiner in terms of selecting conditional statements and checking variable protection.

AntMiner requires all conditional statements and their corresponding validated variables set (VVS). The VVS for each conditional statement contains all variables that are checked directly or indirectly by the conditional statement. However, only conditional statements that control the invocation instance have an effect on explicit checking. Therefore, this section only extracts control dependency conditions related to the given function instance f_i to reduce the computational cost and subsequent matching tasks required for the VVS.

AntMiner uses a recursive method to define the protection of variables in conditional statements. This section uses the non-recursive Algorithm 1 to identify the protected arguments. Procedure ExtractSymbolOfArg (p_{index}) extracts the symbol set $symbolSet_{p_{index}}$ of the argument p_{index} . The symbol set contains variables that constitute p_{index} and all variables in the chain defined by p_{index}. If the intersection of the symbol set and VVS is not empty (line 7), there is an explicit check on argument p_{index}, which avoids recursive operations.

The improved explicit check identification process is shown in Algorithm 1, and consists of three parts: First, the set $conditionSet_{f_i}$ of conditional statements related to the function instance f_i is extracted. Second, the set of the VVS of each conditional statements in $conditionSet_{f_i}$ is calculated. The VVS can be extracted from a backward traversal of the instance statement on the data dependence subgraph (DDS) of the program dependence graph (PDG). Third, each argument is checked to determine if it is protected by a conditional statement. If variable v belongs to the VVS of a conditional statement, or if another variable v' defining v and v' belongs to the VVS, the conditional statement "protects" v. If argument p is "protected" by a control dependency condition in a function instance, there is an explicit check on argument p of the function instance.

Algorithm 1. Identifying the explicit check on function instance

```
procedure GetExplicitCheckInfo($f_i$)
1:  $conditionSet_{f_i}$ = ExtractDependencyCondition($f_i$)
2:  for $c \in conditionSet_{f_i}$ do
3:      $VVS_c$ = ExtractVVS($c$)
4:  for $p_{index} \in argSet_{f_i}$ do
5:      $symbolSet_{p_{index}}$= ExtractSymbolOfArg($p_{index}$)
6:      for $c \in conditionSet_{f_i}$ do
7:          if $symbolSet_{p_{index}} \cap VVS_c \neq \emptyset$ then
8:              $explicitCheckInfo[i][index] = true$
9:  return $explicitCheckInfo$
```

3.2 Implicit check identification algorithm

An implicit check has no obvious single feature compared with an explicit check, and involves the caller setting a valid value for an argument in the form of a direct assignment or an arithmetic operation based on a clear understanding of the function's security specification. In case of a known security specification, one can determine whether the processing of the argument satisfies the corresponding security semantics, thereby decide whether there is an implicit check on the call instance. However, if the security specification is unknown or does not exist, it is difficult to determine whether there is an implicit check on the function instance.

For a certain type of function, there is a constraint relationship among the parameters, such as the function *memcpy (void *dest, const void *src, size_t n)*, and a constraint relationship on *n*, the size of *src* and *dest*. When the function is called, if the constraint is not satisfied, it is likely to cause security problems such as program crash. The constraint is a type of security specification for the function, an implicit check of the validity of its arguments. For this type of specification, this section proposes implicit check identification Algorithm 2, which determines whether there is an implicit check on the function instance by identifying whether a direct or indirect constraint obtains among the relevant arguments.

Algorithm 2 first uses the procedure ExtractAllPath(f_i) to extract all paths from the entry of the caller to the callee f_i, and uses the procedure ExtractDefineChains($path, p_{index}$) to calculate the definition chain of argument (p_{index}) in the execution path ($path$):

$$defChain_{(path,\, p_{index})} = \{(x_1, l_1), (x_2, l_2) \dots\} \tag{1}$$

where (x, l) represents the statement, l defines variable x, and x defines p_{index} along $path$ directly or indirectly. In case of two arguments along the same path, the intersection of their defined chain sets is not empty, indicating that there is a data constraint relationship between them, i.e. there is an implicit check on them.

Algorithm 2. Identifying the implicit check on function instance

```
procedure GetImplicitCheckInfo(f_i)
1:  $executePath_{f_i}$ = ExtractAllPath($f_i$)
2:  for $path \in executePath_{f_i}$ do
3:      for $p_{index} \in paramSet_{f_i}$ do
3:          $defChain_{(path,\, p_{index})}$ = ExtractDefineChains($path, p_{index}$)
4:  for $path \in executePath_{f_i}$ do
5:      if $\exists\, p_s,\ p_t \in parmSet_{f_i},\ s \neq t$ and
            $defChain_{(path,\, p_s)} \cap defChain_{(path,\, p_t)} \neq \emptyset$ then
6:          $implicitCheckInfo[i][s] = true$
7:          $implicitCheckInfo[i][t] = true$
8:  return $implicitCheckInfo$
```

Algorithm 2 is used to identify the implicit check on *memcpy (buf, password, res)* (on line 569) in the code snippet in Fig. 1.

The set of execution paths from the entry of the code snippet to *memcpy (buf, password, res)* is:

$$executePath = \{path_1: 551 \to 552 \to 553 \to 564 \to 566 \to 567 \to 568 \to 569,\ path_2: 551 \to 552 \to 553 \to 564 \to 566 \to 567 \to 569\}. \tag{2}$$

For $path_1$, the definition chains of all arguments are:

$$defChain_{(path_1,'buf')} = \{\} \tag{3}$$

$$defChain_{(path_1,'password')} = \{(password, 553)\} \tag{4}$$

$$defChain_{(path_1,'res')} = \{(res, 568)\} \tag{5}$$

Clearly in $path_1$, there is no non-empty intersection between any pair of definition chains; thus, on $path_1$, the caller imposes no implicit check on the function instance *memcpy(buf, password, res).*

For $path_2$, the definition chains of all arguments are:

$$defChain_{(path_2,'buf')} = \{\} \tag{6}$$

$$defChain_{(path_2,'password')} = \{(password, 553) \tag{7}$$

$$defChain_{(path_2,'res')} = \{(res, 566), (password, 553)\} \tag{8}$$

Obviously in the $path_2$, there is a non-empty intersection between the definition chains of argument, $password$, and the definition chains of argument, res, that is,

$$defChain_{(path_2,'password')} \cap defChain_{(path_2,'res')} = \{(password, 553)\}, \tag{9}$$

Therefore, on $path_2$, the caller imposes an implicit check on the arguments of function instance *memcpy (buf, password, res).*

3.3 Decision algorithm for security-sensitive functions

Algorithm 3. Identifying argument-sensitive function according to check information

```
procedure IsArgumentSensitiveFunction($f, explicitCheckInfo, implicitCheckInfo, \lambda$)
1:  $n$ = GetInstanceNums($f$)
2:  $m$ = GetArgumentNums($f$)
3:  for $j < m$ do
4:      for $i < n$ do
5:          if $explicitCheckInfo[i][j] \cup implicitCheckInfo[i][j]$ then
6:              $checkInfo[i][j] = 1$
7:      $counter[j] += checkInfo[i][j]$
8:      $weight[j] = counter[j]/n$
9:  $sensitiveWeight$ = GetMaxValue($weight$)
10: return ($sensitiveWeight \geq \lambda ? true : false,\ sensitiveWeight$)
```

Algorithm 3 determines whether the objective function is an argument-sensitive function based on the information related to the explicit and implicit checking of all function instances, and the corresponding threshold. In a function instance, if there is an explicit or implicit check on an argument, the instance is considered to conform to a security specification on the argument, and the protection counter of the parameter corresponding to the argument is incremented by one. Then, the protection counter for all parameters and the ratio of the maximum number of parameter protection counters to the total number of instances are calculated, where the latter is called the sensitive measure function:

$$sensitiveWeight = MaxCounter/n \tag{10}$$

where *MaxCounter* represents the largest value of the parameter protection counter, and *n* is the total number of calls to the instance. A high value of the sensitive measure function indicates more instances in the test for argument checking, implies a high probability of the given function being an argument-sensitive function. The threshold of Eq. (10) is set to determine whether the corresponding function is argument sensitive.

4 Experimental results and analysis

To evaluate the effectiveness of the proposed method, 3 well-known open-source projects were chosen for experiments. The detailed information is listed in Tab. 1, where OpenSSL is a cryptography library used for SSL communications, Libtiff is used to manipulate label images, and SQLite is a software library implementing the SQL database engine. All three software have been updated several times, and are widely used mature code projects.

Table 1: Datasets of three open-source projects

Project	Version	Size (M)
OpenSSL	1.1.0f	8.2
Libtiff	4.0.7	8.6
SQLite	3.18.0	74

The argument-sensitive functions mining the results of the three software are shown in Tab. 2. As is evident, our method identified more security-sensitive functions than AntMiner because it considered the implicit check on arguments, which is ignored in AntMiner.

Table 2: The numbers of argument-sensitive functions ($\boldsymbol{\lambda = 0.7}$)

Project	AntMiner	Proposed	Numbers of functions
OpenSSL	2063	2401	6226
Libtiff	575	633	1239
SQLite	1804	2081	4926

The functions mined in the above results were mostly project-specific functions that were unique to the relevant projects, and thus can be only manually confirmed to be argument-sensitive functions. As the number of mined functions was large, it was difficult to accurately evaluate the performance of the proposed method.

To better evaluate the performance of the proposed method, 28 argument-sensitive functions and 32 normal functions of glibc were chosen as benchmark functions for experiments. glibc is a C runtime library released by GUN that is widely used in many software. As the selected functions were commonly used and their security specifications were well known, it was easy to identify the argument-sensitive functions.

The source code of the three open-source software was combined, and AntMiner and the proposed method were used to identify 60 glibc functions. The receiver operating characteristics (ROC) curves of the results are shown in Fig. 3, where the curve of our method is above that of AntMiner. This indicates that our method yielded better performance.

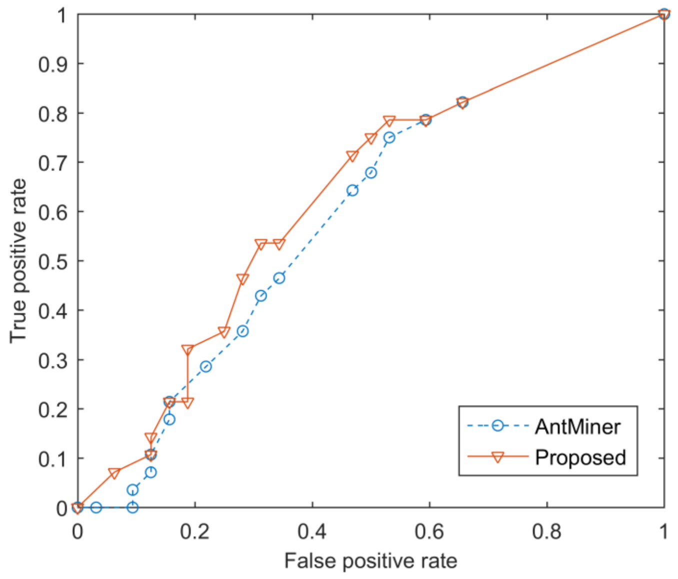


Figure 3: ROC curves of AntMiner and the proposed method

5 Conclusion

The automatic mining of security-sensitive functions is important for improving the efficiency of manual and automated vulnerability detection. This paper analyzed two forms of argument checking and introduced corresponding methods of identification to propose a method to mine argument-sensitive functions. Experiments involving the mining of security-sensitive functions were implemented on large, well-known open-source projects. The experimental results showed that it can outperform AntMiner. However, owing to the different ways of implementing program semantics, the implicit check identification algorithm proposed in this paper failed at times, such as when using a constant instead of a calculation. In future research, we intend to further improve the identification of implicit check to enhance the efficiency of the proposed method to mine security-sensitive functions.

Acknowledgments: This study was supported in part by the National Natural Science Foundation of China (Nos. 61401512, 61602508, 61772549, U1636219 and U1736214), the National Key R&D Program of China (No. 2016YFB0801303 and 2016QY01W0105), the Key Technologies R&D Program of Henan Province (No. 162102210032), and the Key Science and Technology Research Project of Henan Province (No. 152102210005).

References

Cadar, C.; Dunbar, D.; Engler, D. R. (2008): KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. *Proceedings of USENIX Conference on Operating Systems Design and Implementation*, pp. 209-224.

Engler, D. R.; Musuvathi, M. (2004): Static analysis versus software model checking for bug finding. *Proceedings of Verification, Model Checking, and Abstract Interpretation*, vol. 2937.

Ganesh, V.; Leek, T.; Rinard, M. (2009): Taint-based directed whitebox fuzzing. *Proceedings of IEEE International Conference on Software Engineering*, pp. 474-484.

Godefroid, P.; Klarlund, N.; Sen, K. (2005): DART: Directed automated random testing. *Proceedings of ACM Sigplan Conference on Programming Language Design and Implementation*, pp. 213-223.

Godefroid, P.; Levin, M. Y.; Molnar, D. A. (2008): Automated whitebox fuzz testing. *Proceedings of the Network and Distributed System Security Symposium*, pp. 63-79.

Haller, I.; Slowinska, A.; Neugschwandtner, M.; Bos, H. (2013): Dowsing for overflows: A guided fuzzer to find buffer boundary violations. *Proceedings of USENIX Conference on Security*, pp. 49-64.

Kosba, A.; Miller A.; Shi E.; Wen, Z.; Papamanthou C. (2016): Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. *Proceedings of IEEE Symposium on Security and Privacy*, pp. 839-858.

Liang, B.; Bian, P.; Zhang, Y.; Shi, W.; You, W. (2016): AntMiner: Mining more bugs by reducing noise interference. *Proceedings of IEEE/ACM International Conference on Software Engineering*, pp. 333-344.

Luo, X.; Song, X.; Li, X.; Zhang, W.; Lu, J. et al. (2016): Steganalysis of HUGO steganography based on parameter recognition of syndrome-trellis-codes. *Multimedia Tools and Applications*, vol. 75, no. 21, pp. 13557-13583.

Ma, Y.; Luo, X.; Li, X.; Bao, Z; Zhang, Y. (2018): Selection of rich model steganalysis features based on decision rough set α-positive region reduction. *IEEE Transactions on Circuits and Systems for Video Technology*, published online.

Menzies, T.; Greenwald, J.; Frank, A. (2007): Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, vol. 33, no. 1, pp. 2-13.

Microsoft (2017): Microsoft security bulletin MS17-010-critical. https://docs.microsoft.com/en-us/security-updates/SecurityBulletins/2017/ms17-010.

MITRE (2014): CVE-2014-0160. http://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2014-0160.

Özkan, S. (2017): Browse vulnerabilities by date. https://www.cvedetails.com/browse-by-date.php.

Rawat, S.; Jain, J.; Kumar, A.; Cojocar, L.; Giuffrida, C. et al. (2017): VUzzer: Application-aware evolutionary fuzzing. *Proceedings of 24th Annual Network and Distributed System Security Symposium*.

Stallings, W.; Brown, L. (2015): *Computer Security: Principles and Practice*. Prentice

Hall Press, Upper Saddle River, NJ, USA.

Terzi, D. S.; Terzi, R.; Sagiroglu, S. (2015): A survey on security and privacy issues in big data. *Proceedings of 10th International Conference for Internet Technology and Secured Transactions*, pp. 202-207.

Wang, T.; Wei, T.; Gu, G.; Zou, W. (2010): TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. *Proceedings of IEEE Symposium on Security and Privacy*, pp. 497-512.

Wikipedia (2017): WannaCry ransomware attack. https://en.wikipedia.org/wiki/WannaCry_ransomware_attack.

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

Yamaguchi, F.; Lindner, F. F.; Rieck, K. (2011): Vulnerability extrapolation: Assisted discovery of vulnerabilities using machine learning. *Proceedings of the 5th USENIX Conference on Offensive Technologies*, pp. 118-127.

Yun, I.; Min, C.; Si, X.; Jang, Y.; Kim, T.; Naik, M. (2016): APISan: Sanitizing API usages through semantic cross-checking. *Proceedings of USENIX Security Symposium*, pp. 363-378.

Zhang, Y.; Qin, C.; Zhang, W.; Liu, F.; Luo, X. (2018): On the fault-tolerant performance for a class of robust image steganography. *Signal Processing*, vol. 146, pp. 99-111.