Target function location based on code coverage analysis

Kaiwen Jin*, Qingbao Li, and Zhifeng Chen

State Key Laboratory of Mathematical Engineering and Advanced Computing, Henan, China

Abstract. How to locate the target function faster and more accurately is a key problem of Automatic Reverse-Engineering of Software Programs. In order to solve this problem, a target function location method based on code coverage analysis is proposed. Firstly, it obtains function call information of the program and calculate the suspicious rate of each function. And then a stability factor is proposed to reduce the noise functions. Finally, the target function is successfully located. The experimental results show the method proposed has linear time complexity. For software programs with million function calls, it can accurately locate the target function within several minutes, the performance and accuracy are greatly improved compared to the contrast methods.

1 Introduction

In the process of Software Reverse-Engineering, the first job is to determine the cut-in point of the reverse analysis. Usually, the function containing cut-in point is regarded as the target function. How to locate the target function faster and more accurately is an important guarantee for the efficient completion of reverse analysis. In recent years, some automatic location methods have been developed, including difference set calculation, API sequence alignments, dependency analysis, and functional feature recognition.

Based on the method of difference set calculation, Renieris M et al [1] proposed "nearest neighbor" model which is locating the target position by calculating the difference set of the execution paths. Wang et al [2] executes the program $1+N(N \in \mathbb{N}^+)$ times and find the target function by clustering the difference set. Huang et al [3] filters the redundant execution paths by clustering and improve the locating efficiency. But this kind of method has a lot of noise function.

Based on the method of API sequence alignments, Ye et al [4] proposed a method of generating harmonic sequence, which avoided the "blind area" caused by nearest neighbor path selection and reduced the space of subsequent search. But this kind of method had higher time complexity than difference set calculation.

Based on the method of dependency analysis, Chen et al [5] proposed using DG (Dependence Graph) method to analyze function call dependency and data dependency of program, and programmer is able to search DG to locate the feature code. Zhang et al [6] proposed CP(Capture Propagation) method to establish the CFG(Control Flow Graph), it

* Corresponding author: 394884987@qq.com

can locate target code by calculating the suspicious rate of every edge. Baah et al [7] proposed PPDG(Probabilistic Program Dependence Graph) statistics conditions between program procedure and entity, and compare the differences of cases to locate the target module. There are similar methods by analyzing the dependence [8], [9] like program slice technology.

Based on the method of feature recognition, AD Eisenberg et al [10] sort the characteristic correlation of function for program execution trace to locate feature code; Xie et al [11] use the working mechanism and encapsulation characteristics of the MFC(Microsoft Foundation Classes) framework to quickly locate the message response function of the MFC program. In addition, there are many studies on encryption algorithm identification based on feature [12], [13]. This kind of method is very good for identifying specific function, but not universal.

Aiming at shortcomings of the above methods, we propose a method of code coverage analysis to locate target function. It makes whole locating process reach the linear time complexity and shorten the locating time. In addition, we propose the stability factor according to the modular design characteristics of the program. It reduces the interference of the noise function, and improves the accuracy of the location range. For our method is only related to the function call path, it is universal.

2 Principle

According to the location principle, that means, finding the closest trace from the trace set without target function to compare with trace containing target function, ideally, the different part of the trace is the target function.

First, trigger the target function and run the program, we will get a function call trace by using DBI (Dynamic Binary Instrumentation). Then, do not trigger the target function, run the program several times according to the previous operation and get the corresponding function call traces. It is assumed that each function may be target function, and the function appearing on the trigger path is more likely to be target function. We statistics the coverage times of all the function nodes in CG (Call Graph), and calculate the suspect probability of each function node according to the designed suspicious rate formula. From the perspective of probability analysis, we select the candidate list of the target function. According to the location principle, the target function must exist in the candidate list, but there are also some noise functions. In order to filter noise functions, the stability factor is designed to reduce the influence of the noise function on target function location. Finally, all the locating process is shown in figure 1.

Fig. 1. Location flow chart.

3 Acquisition of program execution information

By analyzing the program execution information, we can obtain the control flow and data flow of the program. And DBI is the method that is transparent for monitored program, and does not affect the program running results. We will use DynamoRio [14] to obtain information of program.

Define 1 Function Call $f, f : \alpha \rightarrow b$ means function a is caller, and function b is callee. a, b are the names of function.

Define 2 CG (Call Graph), expressed in $G<V,E>$, V is node set of G . And E is edge set of G . $E = \{e(x,y,depth,od) | e \in E, depth, od \in Z^+\}$, x is the set of predecessor nodes, and y is the set of successor nodes. $depth$ is the minimum number of nodes that from the initial node e passes through. od represents the out degree of the node, that is the size of the node set y .

Fig. 2. A sample of CG.

The relationship between the source code and the CG is shown in figure 2. The main function calls the function f_1 first, then calls the f_2 or f_3 according to the condition, and finally calls f_4 . However, this is just a static CG based on source code. Actually, in the execution of the program, part of the code may not be run, and the functions in this part will not appear in the CG. For example, if condition is true, the function f_3 is not executed, so there is no f_3 node in the dynamic CG.

Define 2 FCP(Function Call Path), in CG, each node is a function of the program, the program from the initial node state transfer to the end node state will form a node sequence i , called FCP, record as $W_i, W_i = (e_{i_0}, e_{i_1}, ..., e_{i_n})$.

In order to get the FCP of the program at runtime. When the program is running, the following three kinds of instructions need to be processed:

- (1) *call* instruction. Function identification needs to match the *call* instruction and *ret* instruction. When program meet *call* instruction, only do push operation. Meanwhile, we record the instruction register ESP's value and push the address of the next instruction into the shadow stack. If the target instruction is in system DLL space, set Flag = true;
- (2) *ret* instruction. If the current value of ESP is equal to the value of ESP which on the top of the shadow stack, the stack meet the balance and the function is identified, record the function information. Otherwise, we need to check if the return value of program's stack is equal to the *ret* instruction's target address. If they are equal, stack imbalance function is identified, if they are not equal, the *ret* instruction is judged to an ordinary jump instruction.

If the destination address is in system DLL space, set Flag=true;

(3) other instructions. When the first instruction of trace is not *call* or *ret*, if Flag=true, we first set Flag to false, and check if the return value on the top of the shadow stack is equal to the current instruction’s address, if they are equal, API call is identified. Otherwise, callback function is identified, and then push the current instruction’s ESP and the ESP’s target address into the shadow stack.

The pseudo code of CG generating algorithm is as follows.

Algorithm 1: FunctionCallGraphGeneration
Input: Program Instruction Stream Output: Function Call Graph Declaration: ShadowStack //used to recognize function
BEGIN: <i>initialize ShadowStack ;</i> while <i>not end of Program</i> do <i>get an instruction from Program Instruction Stream;</i> if <i>instruction is call</i> then <i>push ESP and address of the next instruction;</i> if <i>instruction’s target address is in System Dll Space</i> then <i>Set flag to true;</i> end if else if <i>instruction is ret</i> then <i>pop ESP;</i> if <i>ESP match current instruction’s ESP</i> then //we can use Graphviz to draw FCG, information including length and od <i>recognize function and record its information;</i> if <i>instruction’s target address is in System Dll Space</i> then <i>Set flag to true;</i> end if end if else // other instruction if <i>instruction is the first instruction of Trace and flag is true</i> then <i>set flag to false;</i> <i>pop address;</i> if <i>address match current instruction’s address</i> then <i>recognize API;</i> else <i>push current instruction’s ESP and data which saved in ESP;</i> end if end if end if end while END

4 Covering Analysis

Define 4 the set of FCPs: in software testing process, there is a relationship between the program *P* and test cases *T*, record as *W_T*:

$$W_T = (P,T) \tag{1}$$

The program $P=\{f_1, f_2,..., f_n\}$ are used to locate the target function, and input test cases set $T=\{t_1, t_2,... t_m\}$ into program P one by one. According to whether the test case trigger target function or not, we can get a result set $F=\{r_1, r_2,..., r_m\}$, and r_i ($i=1, 2,..., m$) is Boolean type. When r_i is True, it means target function is triggered, conversely, means target function is not triggered:

$$W = \{W \mid r = True\}$$

(2)

$$W_{TN} = \{W_i \mid r_i = False\}$$

(3)

Suppose that each function in the program P is likely to be target function, then the approximate degree between each function and target function is called the suspicious rate of the function, which is denoted as $S(f)$. $S(f) \in [0,1]$, and the greater the value is, the more likely the function f is target function f^* .

For test cases set T , we can get the W_T by DBI. Then we can statistics the total number of the function e is called, record as $N(e)$. There is the following relationship:

$$N(e) = N_p(e) + N_f(e)$$

(4)

$N_p(e)=N(e|e \in W_{TP})$ means the total number that function e in the test cases set which target function is triggered, $N_f(e)=N(e|e \in W_{TN})$ means the total number that function e in the test cases set which target function is not triggered, and there:

$$N_f(f^*)=0$$

(5)

For each function node e , there is a two-element feature $e(Executed, W_T)$, where Executed is the Boolean type, $W_T \in \{W_{TP}, W_{TN}\}$. We can get a binary set $e'=(e_1', e_2', e_3', e_4')$, there are:

$$e_1' = (0,0), if \text{ Executed } = false, W_T = W_{TN}$$

(6)

$$e_2' = (0,1), if \text{ Executed } = false, W_T = W_{TP}$$

(7)

$$e_3' = (1,0), if \text{ Executed } \neq false, W_T \neq W_{TP}$$

(8)

$$e_4' = (1,1), if \text{ Executed } \neq false, W_T \neq W_{TN}$$

(9)

For the two-element feature $e(x, y)$, if $x=1$ and $y=1$, a_{ep} means the number of test cases that trigger target function when the function node e is executed. if $x=1$ and $y=0$, a_{ef} means the number of test cases that do not trigger target function when the function node e is executed. if $x=0$ and $y=1$, a_{np} means the number of test cases that trigger target function when the function node e is not executed. if $x=0$ and $y=0$, a_{nf} means the number of test cases that do not trigger target function when the function node e is not executed. The relationship between them are shown in table 1.

Table 1. Relationship between function node and test case.

		x		
		1	0	sum
y	1	a_{ep}	a_{np}	$ W_{TP} $
	0	a_{ef}	a_{nf}	$ W_{TF} $
	sum	$a_{ep} + a_{ef}$	$a_{np} + a_{nf}$	m

According to $S(e)$, a suspicious candidate function list can be generated. We sort the suspicious functions in the candidate list, and finally find target function. In particular, the pseudo code of the code coverage analysis algorithm is shown as follows.

Algorithm 2: CoverageAnalysis**Input:** CallGraph, hash_map< t_i >, $i=1, 2, \dots, m$ **Output:** $S(e)$ **Declaration:**EP — Number of test samples that execute f^* when function node e is executed;NP — Number of test samples that execute f^* when function node e is not executed;EF — Number of test samples that don't execute f^* when function e is executed;NF — Number of test samples that don't execute f^* when function node e is not executed;**BEGIN:** for each e_j in CallGraph do for $i = 1 \rightarrow k$ do // when $i = 1, 2, \dots, k$. $W_T = (P, T) = W_{TP}$ $EP += \text{hash_map}\langle t_i \rangle.\text{find}(e_j)$;

end for

 $NP = k - EP$; for $i = 1 \rightarrow s$ do // when $i = 1, 2, \dots, s$. $W_T = (P, T) = W_{TN}$. And $k + s = m$. $EF += \text{hash_map}\langle t_i \rangle.\text{find}(e_j)$;

end for

 $NF = s - EF$;

$$S(e_j) = \frac{EP}{EP + NP} \times \left(\frac{EP}{EP + NP} + \frac{EF}{EF + NF} \right)^{-1};$$

end for

END

The time complexity of algorithm is $O(k*n) + O(s*n)$. The sum of k and s is a constant, record as m . So the final time complexity is $m*O(n)$, it is a linear complexity.

5 Stability Factor

Actually, modular software design concept makes the functional modules have good independence, the function division is very clear, the public module is more frequently reused. Based on such characteristics, we propose stability factor to describe the stability of software function modules, and find target function from candidate function list.

We expect functional module to have less impact on other modules, which requires functional module to have strong independence. Therefore, the functions which in module should have a larger value of *depth* and smaller value of *od*. It is similar to the leaves and nearby nodes which are deep in the Call Tree structure. They are different from other functions, in the whole process of software execution, the influence on other functions is relatively small, which satisfies the independence of functional modules. On the other hand, target function usually is not leaf node, but the public function is of the reuse rate very high. According to the call number, we can filter this kind of noise function, reduce the size of the candidate function list. Stability factor is given as follow:

$$\alpha = \frac{\text{depth}}{\text{od} + 1} \times N(e)^\theta, \theta \in (-1, 0) \quad (10)$$

6 Experiment

6.1 Functional testing

This section gives a detailed test of the WinRAR3.7 Beta version. WinRAR provides the window interactive interface and command line interface. In this paper, we will use command line interface to locate the original encryption function in WinRAR.

First of all, close the ASLR (Address Space Layout Randomization), and let the software perform 1 encryption operation and 3 compression operations, the operation object is two *.txt format file. The function call information of WinRAR as shown in table 2.

Table 2. Operation information of WinRAR.

	Trigger target function	Do not trigger the target function		
		First run	Second run	Third run
Number of functions	503	500	500	500
Total number of function call	1262901	12313	12308	12591

After the function coverage statistics, calculate the suspicious rate $S(f)$ for each function, and set threshold to 0.75, the function whose $S(f)$ is greater than 0.75 will enter the candidate list, Finally, a candidate list with 7 elements is obtained, as shown in table 3.

Table 3. Candidate list of WinRAR.

Ordinal	Address	Numbers of call
1	40b604	1
2	40ced0	7
3	40ce6c	2
4	40d088	2
5	40d334	7
6	40c2a8	7
7	42b9e0	1

Using DBI to get the function call information to build CG. Calculate stability factor for 7 candidate functions and sort them (There $\theta = -0.5$). The result table 4 shows.

Table 4. Location result of WinRAR.

Ordinal	Address	<i>od</i>	<i>depth</i>	α
1	40b604	0	5	5
2	40ced0	1	6	1.13
3	40ce6c	2	5	1.18
4	40d088	0	6	4.24
5	40d334	0	7	2.65
6	40c2a8	1	6	1.13
7	42b9e0	24	2	0.08

According to the results of sorting, we can judge the encryption function is 0x40b604 or 0x40d088, using Ollydbg to further analysis we can find that sub_40b604 is responsible for password import for file encryption, and sub_40d088 run after sub_40b604, it is target function that implements file encryption.

In order to verify the versatility and comprehensiveness of this method, we select more different types of software to test, and the specific test list is shown in table 5.

Table 5. Test list.

Software	Open/non-open source	Main function of software	Target function
Notepad	Non-open source	File editing	Save file function
Imagestone	Open source	Image processing	Function of converting RGB image into gray image

Greenvpn	Non-open source	VPN	VPN connection function
Filezilla Client	Open source	FTP file sharing	FTP server connection funcion
Auditor	Non-open source	Security and secrecy inspection	Internet trace search function
FeiQ 2013	Non-open source	Instant messaging in LAN	Send message function
Netcat	Open source	Network security tool	Send file function
Hash	Non-open source	Calculating and checking the MD5 of software	Calculating MD5 function

The coverage analysis method is used to locate target function of each test case, and the experimental results are shown in table 6.

Table 6. Location results of test software programs.

Software	Number of functions	Total number of function call(million times)	Size of candidate list	The α ranking of target function	Address of target function
Notepad	42	30.4	5	1	0x01004b99
Imagestone	1465	550.5	10	1	0x0040e010
Greenvpn	536	129.5	19	3	0x0040f4e0
Filezilla Client	5666	489.2	204	27	0x00dd5bf0
Auditor	893	591.9	49	2	0x004538f0
FeiQ 2013	6297	803.3	291	14	0x00408dc0
Netcat	73	12.7	4	1	0x004056e0
Hash	39	219.8	6	1	0x00402af0

From table 6, in general, the larger a software, the larger the candidate list. When the interaction between user and software becomes more complex, leading to an increase in the difference between W_{TP} and W_{TN} , and the candidate list becomes larger. However, the stability factor can further filter the candidate list and reduce the number of noise functions. For example, the software Filezilla Client, its candidate list has 204 functions, but its stability factor of target function only ranks 27.

6.2 Performance testing

Experimental environment: Windows7 SP1 64 bit, Intel (R) i7-4710MQ CPU@2.50G Hz, memory 4.00G.

The effect of DBI on the program running time is analyzed by recording the free running time and the running time under DynamoRio in each test case, and the results are shown in figure 3.

Fig. 3. Influence of instrumentation on program’s running time.

In figure 3, The multiplier of the vertical axis data represent the extension of the running time. Generally, the influence of DBI on running time is less than 10 times, the total run time is only a few minutes, it can satisfy the actual locating needs.

The following are the experimental results and comparison of our methods and other locating methods. The result is shown as figure 4.

Fig. 4. Comparison of location algorithms.

The upper picture is our method, its time overhead satisfies the linear time. The other picture is a comparison of the three methods, we can see that both our method and difference set calculation satisfy the linear time. But the latter exist a lot of noise functions. And the time complexity of API sequence alignments is $O(n_1 * n_2 * \dots * n_i)$, where i is the number of sequences. Dependency Analysis has a higher time complexity due to the need to parse the dependencies between elements.

7 Conclusions

Experiments show that our method can determine the candidate list in linear time, and the stability factor can reduce the noise functions. But this method has some disadvantages, such as selection of test cases, how to trigger target function or how to avoid triggering target function, this is a problem worthy of further study. In addition, when the interaction between user and program is too complex, the candidate list is too large, there still a lot of

work to be done even though calculating stability factors. In view of the above problems, how to achieve a more rapid, precise target function location method need further research.

Acknowledgment

This work is supported by the National Social Science Foundation of China (No. 15AJG012).

References

1. Renieris M and Reiss SP 2003 Fault Localization With Nearest Neighbor Queries *Proc. Int. Conf. on Automated Software Engineering(Montreal)* vol 2003 (IEEE) pp 30-39
2. Kang F, Wang Q, Xiao YN and Huang HJ 2013 Research on key functions locating technique based on software execution trace difference comparison *Journal on Communications* **34** chapter 9 pp 177-184
3. Huang XH 2017 Fault Localization Method Based on Path Clustering *Software Guide* **16** chapter 3 pp 3-6
4. Ye JM, Xie Q, Jiang L, Li SS and Xu L 2011 Generation of a Consensus Sequence and its Applications in the Fault Location *Computer Science* **38** chapter 3 pp 162-165
5. Chen K 2010 Case Study of Feature Location Using Dependence Graph, after 10 Years *Int. Conf. on Program Comprehension(Braga)* vol 2010 (IEEE) pp 1-3
6. Zhang ZY, Chan WK, Tse TH, Jiang B and Wang XM 2009 Capturing propagation of infected program states *The Joint Meeting of the European Software Engineering Conference and the ACM Sigsoft Symposium on the Foundations of Software Engineering(Amsterdam)* (ACM) pp 43-52
7. Baah GK, Podgurski A and Harrold MJ 2009 The probabilistic program dependence graph and its application to fault diagnosis *Software Engineering IEEE Transactions on* **36** chapter 4 pp 528-545
8. Wen WZ and Chen S 2015 Concurrent Sequence Slice *Scientific and Technological Innovation* **35** pp 15-16
9. Teng QY 2016 Fault Localization Based on Dynamic Program Slicing *Nanjing University of Posts and Telecommunications*
10. Eisenberg AD and Volder KD 2005 Dynamic feature traces: finding features in unfamiliar code *Int. Conf. on Software Maintenance(Budapest)* vol 2005 (IEEE) pp 337-346
11. Xie YM, Shu H, Chen JM and Xiong XB 2009 Location of MFC messages processing functions in converse analysis *Journal of Computer Applications* **29** chapter 5 pp 1393-1396
12. Oprisnik A, Hein D and Teufl P 2016 Identifying Cryptographic Functionality in Android Applications *Int. Conf. on Security and Cryptography(Vienna)* vol 2016 (IEEE) pp 151-162
13. Calvet J and Marion JY 2012 Aligot: cryptographic function identification in obfuscated binary programs *ACM Conference on Computer and Communications Security(Raleigh)* (ACM) pp 169-182
14. Derek LB 2004 Efcient, Transparent, and Comprehensive Runtime Code Manipulation *Massachusetts Institute of Technology*