Function exclusion in automated security patch testing using chopped symbolic execution

1st Phan Ninh Thai
Department of Information Security
Le Quy Don Technical University
Hanoi, Vietnam
phan123123@gmail.com

2nd Hung Nguyen Viet
Department of Information Security
Le Quy Don Technical University
Hanoi, Vietnam
hungnv@lqdtu.edu.vn

3rd Nathan Shone
School of Computer Science & Mathematics
Liverpool John Moores University
Liverpool, United Kingdom
n.shone@ljmu.ac.uk

4th Mikhail Babenko
Institute of Mathematics and Information Technologies
North-Caucasus Federal University
Stavropol, Russia
mgbabenko@ncfu.ru

***Abstract*—Patch testing is a core component of patch management and is used to verify that modified software modules (i.e. an update or patch) work as expected (functional testing) and do not contain any known vulnerabilities (security testing). Security patch testing requires a lot of time and a professional security knowledge from the tester. In recent years, chopped symbolic execution has been successfully applied in automatic or semi-automatic program testing, to reduce the amount of testing work. Chopped symbolic execution (Chopper) allows users to specify "uninteresting" functions to ignore during analysis, therefore allowing the testing of software modules without running all functions of the program. It is an effective solution for path explosion (one of the main problems of symbolic execution). The effectiveness of the chopped symbolic execution method in patch testing depends on how well the ignored functions are initially chosen. In this paper, we propose a novel method to automatically exclude functions for chopped symbolic execution in patch testing, using a control flow graph. Moreover, we use cyclomatic complexity to optimize the speed of the testing process. Experimental results show that our method can automatically choose the ignored functions and reduce the required testing time, in comparison to typical Chopper techniques.**

***Index Terms*—Security patch testing, Symbolic execution, Chopper, control flow graph**

I. Introduction

Developers need to continually review software and create patches to address identified bugs, which can be a labor-intensive and error-prone process. Users are often reluctant to upgrade their software to the most recent version because they're not sure whether the updated functions are safe [1]. Moreover, code patches have the potential to introduce or cause new vulnerabilities [2], [3] partly because developers can't find all of the bugs. Therefore, a method that can automatically find the vulnerabilities in the new patch is necessary.

Various techniques have been proposed for use in automatic security patch testing. For example, in [4], the authors present an automatic patch-based vulnerability and fuzzing method. It specifically focuses on a privilege elevation vulnerability and fuzz tests to identify unknown bugs. Two-way taint analysis techniques are used to measure the relevance among address, object, operation semantic and constraint features. However, the fuzzing method often has a low efficiency because of missing too many possible input testing cases.

Symbolic execution is a powerful technique for finding software vulnerabilities. Many tools were introduced to find software vulnerabilities [5]–[11]. In theory, this technique can solve the coverage problem of fuzzing, but it still has limitations in practice. Symbolic execution often struggles to reach deep parts of the code due to the well-known path explosion problem and constraint solving limitations [12]. For security testing, there are some existing solutions using symbolic execution techniques, such as KLEE [5]. KLEE is one of the most popular symbolic execution tools. It is a symbolic virtual machine built on top of the LLVM compiler infrastructure [5].

There is existing research on the use of KLEE for patch testing - KATCH [13] but the authors primarily focus on coverage during testing. There are great solutions for choosing input or identifying symbolic execution techniques via static analysis. However, the problem we want to focus on is path explosion. KLEE and other symbolic execution tools are used for program testing, but in larger programs it doesn't yield good results. The reason for this is the path explosion problem. Most methods based on the symbolic execution approach have to test all paths of a program. Therefore, these methods are influenced by the path explosion problem, one of the biggest challenges of symbolic execution. Chopped symbolic execution [12] was introduced to solve this problem with Chopper - an extended version of KLEE. Authors proposed the mechanism to define the functions that can be temporarily ignored during analysis. This mechanism is very useful for the patch testing methods that use symbolic execution because it omits many unnecessary paths from analysis, thus helping to solve the path explosion problem.

Chopper does not actually ignore the excluded functions,

978-1-6654-1001-4/21/$31.00 ©2021 IEEE

as this may lead to both false positives and false negatives in finding bugs. In fact, the excluded parts are executed lazily, whereby their effects may be observable by code under analysis. Chopped symbolic execution leverages various on-demand static analyses at runtime to automatically exclude code fragments while resolving their side effects [12]. Although chopped symbolic execution is a great idea, it has one major weakness in automatic patch testing - the excluded functions have to be determined in advance by the tester. To identify whether a function is of interest, static source code analysis is required, thus reducing the advantage of an automatic analysis tool like Chopper.

Therefore, instead of using expert knowledge to determine the uninteresting functions, our approach intends to enhance Chopper by automating the static code analysis used to detect uninteresting functions. In patch testing, the commits bring the information about modified code. The modified functions or additional functions can be determined easily with a version control or source management system [14]–[18]. However, finding out which functions are not affected by an update is not easy. It will take a long time to analyze the source code. Moreover, if too many functions are excluded, Chopper will spend a lot of time finding side effects, taking snapshots and recovering when needed.

In this paper, we propose a novel function exclusion method called *chop's automatic exclusion* that can determine suitable functions that will be excluded with the target function as input for chopped symbolic execution in patch testing. The target function is a function that can be modified or added in this patch. We evaluate the effectiveness of this method using a prototype implementation and present the results obtained, which demonstrate its effectiveness in discovering memory vulnerabilities when patch testing with Chopper.

Our main contributions in this work are as follows:

1. Proposed a novel method to exclude functions **automatically** for chopped symbolic execution in patch testing (chop's automatic exclusion).

2. Built a prototype implementation of our technique as a dynamic library of opt, a modular LLVM optimizer and analyzer [19] (that we called ChopperAEx).

II. Overview of Choosing the Excluded Functions

In this section, we give an overview of choosing the functions that should be excluded in patch testing with chopped symbolic execution, and how to get the best performance. An example using a simple program is shown in Fig. 1 and Fig. 2. Supposing that *f* is the function which was modified in the patch, our aim is to test function *f* with the program's input (we also call function *f* - the target function). Fig. 1 shows a part of the pseudocode of the program (function main and test) of a sample program. Fig. 2 shows the Control Flow Graph (CFG) of the program and the functions will be excluded (function *n* and function *m*).

Starting with the function main, we have a symbolic value as a parameter which is α. For being called the target function, the path is from *main* to *test* then *m* and *f* at the end, as long as we have str(α) equal "OK" condition. It is easy to decide that function *n* should be excluded because it is not in the path which leads to function *f*. If function *n* is excluded, the symbolic execution engine will not waste time and resources for exploring the useless path from function *n*. However, function *m* is not that easy. According to the program's flow, if we want to execute function *f*, function *m* will be executed first. But we can still exclude function *m* in case all pointers in function *f* are not affected by function *m*. In simpler terms,

```
void test(char* a){
  char* str1 = "OK";
  if(str(str1) == str(a)){
    m();
    f(a);
    }
  else{
    n();
  }
}

int main(int args, char** argv){
  int x,y;
  char* a = argv[1] //symbolic
  test(a);
}

void f(char *a){
...
}
```

Fig. 1. Pseudocode of the example program.

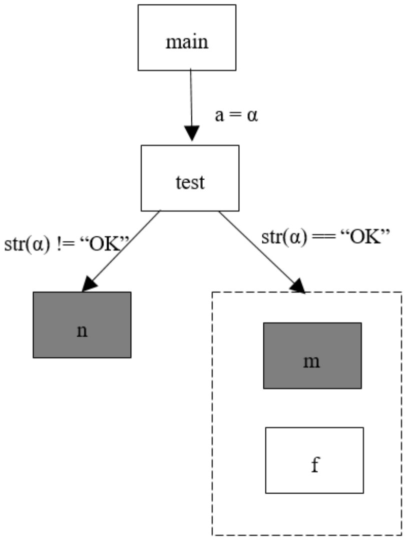

Fig. 2. Illustration of the selection excluded functions with chopped symbolic execution on the example

if no variable in function *f* is modified by function *m*, we will exclude function *m*. However, it is not simple like that.

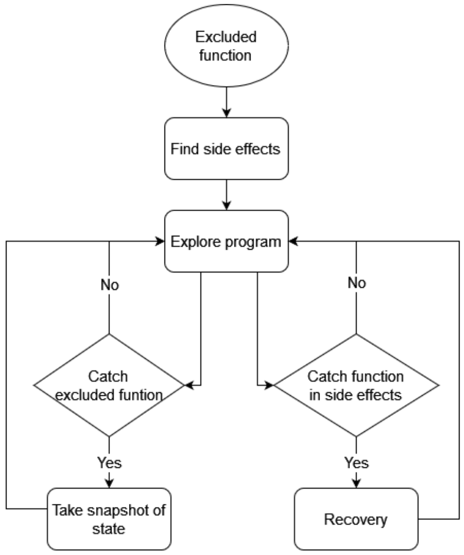

Fig. 3. Main process of chopped symbolic execution

Fig. 3 shows the main process of chopped symbolic execution. We can see that even excluded functions take an amount of time for finding side effects. But it is performed by the static analysis technique [12], so it does not matter. However, taking snapshot task and recovery task are not like that. They take a long time. In some cases, exploring a function maybe faster than taking a snapshot if this function has few branches. Not to mention recovering when catching a function in side effects or taking a snapshot multiple times in branched cases (like comparing with if), they will take more time and reduce performance. Back to our example, if function *m* is "simple", we will not choose it for exclusion and vice versa. We suppose that function *m* is a "simple function" (will be explained in the following section) and we can exclude it for increasing performance.

The above example showed the effectiveness of the excluded functions method for patch testing with chopped symbolic execution (Chopper). Nevertheless, using Chopper tester need to analyze the program and use the expert knowledge to determine which function affects the target function and should be excluded. It is the limitation of chopped symbolic execution in patch testing that we mentioned in the previous section. To address this problem, we propose a method that can automatically choose the excluded function. Our method will be described in more detail in the following section.

III. Chop's automatic exclusion

In this section, we will present the target algorithms that we used in the proposed method (*chop's automatic exclusion*). Algorithm 1 presents the key step to choose excluded functions. This algorithm operates on a simple imperative C-like application. Firstly, our method will find the shortest path to the target function using the Breadth First Search (BFS) algorithm with the GETSHORTESTPATH function one line 1. Secondly, we walk through the shortest path to target function and try to find all other paths which lead to the target function via the loops at line 3–5. Thirdly, we checked if the function created symbolic values or not by list *sigFunctions*. Before that, we examine the functions that the users define by using the IS_VALUABLE function. Here, a list of standard libraries is used as a whitelist to determine which will be chosen for the next step. Finally, we choose functions that have a cyclomatic complexity (calculated by Algorithm 2) larger than the predefined complexity threshold. We scrutinize the basics block and instructions in paths to find out the position of the excluded function's callers from the debug information.

Algorithm 1 Chop's automatic exclusion

```
Input: target
Output: excludedFuntions
 1: $shortestPath \leftarrow GETSHORTESTPATH()$
 2: $excludedFuntions \leftarrow \emptyset$
 3: for $function$ in $shortestPath$ do
 4:   for $basicBlock$ in $function$ do
 5:     for $instruction$ in $basicBlock$ do
 6:       $f \leftarrow GETFUNCTION(instruction)$
 7:       if $f$ in $shortestPath$ then
 8:         continue
 9:       else if $f$ in $sigFuntions$ then
10:         $excludeFuntions \leftarrow \emptyset$
11:       else if not $IS\_VALUABLE(f)$ then
12:         continue
13:       else if $CYCLOMATICCOMPLEXITY(f) < MIN\_COMPLEXITY$ then
14:         continue
15:       end if
16:       $line \leftarrow DEBUGINFO(f)$
17:       $excludedFuntions \leftarrow (excludedFuntions \cup \{(f, line)\})$
18:     end for
19:   end for
20: end for
21: return $excludedFuntions$
```

The cyclomatic complexity of a structured program is defined with reference to the CFG of the program. CFG is a directed graph that its nodes are the basic blocks of the program. Two nodes in CFG have an edge if they call each other inside their function [20]. The complexity M is calculated by the formula:

$$M = E - N + 2P \qquad (1)$$

Where E: number of edges
N: number of nodes
P: the number of components

P of a function's CFG or a program's CFG is always equal to 1, so we have a simpler formula is:

$$M = E - N + 2 \qquad (2)$$

The cyclomatic complexity of a function is the number of linearly-independent paths which go through it. The time required to examine a function in symbolic execution depends on the number of function paths. According to the mechanism of chopped symbolic execution in Section II, when an excluded function is determined, chopped symbolic execution must find side effects of that function via pointer analysis. Moreover, it must take a snapshot of the current state when catching that function. This process is time consuming, and in some cases it takes longer than exploring the same function using dynamic symbolic execution techniques. Not to mention recovering step of function's state if catching that function again afterwards, it also takes more time and this time also depends on number of paths which go through that function. The cyclomatic complexity threshold helps remove simple functions in the excluded list. Therefore, we use cyclomatic complexity for filtering excluded functions to improve performance.

Algorithm 2 Calculating the function's cyclomatic complexity CYCLOMATICCOMPLEXITY

```
Input: function
Output: complexity
$result \leftarrow 0$
if $GETNAME(function) == target$ then
    return $-\infty$
end if
$iteratorStatus \leftarrow FIND(statusMap, function)$
$iteratorComplex \leftarrow FIND(complexMap, function)$
if $iteratorStatus == NULL$ then
    $statusMap = statusMap \cup \{(function, HOLD)\}$
    $complexMap = complexMap \cup \{(function, 0)\}$
else
    if $iteratorStatus \rightarrow second == HOLD$ then
        return 1
    else if $iteratorStatus \rightarrow second == DONE$ then
        return $iteratorComplex \rightarrow second$
    end if
end if
$result \leftarrow result + INTERNALCOMPLEX(function)$
for $calledFunc$ in $function$ do
    if $IS\_VALUABLE(calledFunc)$ then
        $result \leftarrow result + CYCLOMATICCOMPLEXITY(calledFunc)$
    end if
end for
return $result$
```

Algorithm 2 presents the method for calculating the function's complexity. There are two things that need to be noticed in Algorithm 2. The first one is the INTERNALCOMPLEX function that calculates the complexity with the internal basic blocks inside it. The second one is the recursive algorithm to calculate complexity from the edges of CFG which lead to the other functions.

Our implementation works with LLVM bitcode [21], a dynamic plugin of the LLVM optimizer - opt [19], which is called ChopperAEx. The result from ChopperAEx will be used by Chopper as the "–skip-functions" argument. It contains the functions' names and line number where the functions are called. The plugin was compiled by LLVM 3.4 [22] and CMake version 3.2.2 [23].

IV. Evaluation

For evaluating the proposed method, we reproduced 6 failure scenarios using information derived from the same CVEs as used in [12]. Comparing between ChopperAEx's automated and Chopper's manual expert's knowledge-based methods for finding "skip functions" demonstrates the possibility of chop's automatic exclusion in practice. Moreover, when comparing between KLEE and Chopper with ChopperAEx, we identified a significant improvement on resolving the path explosion problem of symbolic execution techniques in patch testing.

Table I shows the time of finding out the bugs. We used the computer with the OS Ubuntu 14, 4GB of RAM and an i3-8100 @ 3.6GHz CPU (only use 1 core 2 threads). All cases were created as execution drivers for *libtasn1* library to exercise the library from its public interface, simulating the interactions of an external program (e.g., GnuTLS) [12]. Table I shows the time for detecting the vulnerabilities when using KLEE, Chopper with the same setup used by its authors and Chopper with ChopperAEx to find excluded functions automatically. ChopperAEx uses the function which leads to the bug directly as the target function and defines 5 as the complexity threshold in all cases. ChopperAEx's output used as the "–skip-functions" argument of Chopper. In the Table I, timeout means the executing time is over 3 hours and *Error* means an error occurred during the runtime of Chopper.

TABLE I
Detailed results for the failure reproduction experiments

CVE	Search Algorithms	KLEE	CHOPPER	CHOPPER with ChopperAEX
		Time(s)	Time(s)	Time(s)
2012-1569	Random	499.94	125.11	16.43
	DFS	99.91	27.56	4.96
	Coverage	291.72	136.68	10.88
2014-3467(1)	Random	Timeout	487.70	12.12
	DFS	111.87	7.41	5.32
	Coverage	Timeout	195.45	8.30
2014-3467(2)	Random	2.29	579.33	67.29
	DFS	Timeout	159.07	Error
	Coverage	2.09	495.18	41.23
2014-3467(3)	Random	Timeout	426.40	Timeout
	DFS	Timeout	13.90	Timeout
	Coverage	Timeout	259.65	Timeout
2015-2806	Random	281.86	142.68	225.22
	DFS	7733.60	584.82	5990.59
	Coverage	210.33	54.84	171.25
2015-3622	Random	Timeout	64.37	18.03
	DFS	Timeout	1123.22	20.54
	Coverage	Timeout	69.59	18.18

As shown in Table I, Chopper with ChopperAEx is able to complete it's operations much quicker than the original

Chopper with author's setup in 4 of the 6 test cases. For the CVE 2014-3467(3), we received 3 timeouts when using Chopper with ChopperAEx's output because the target function was suddenly called after the main function. In these cases, the target function is located right after main function so ChopperAEx doesn't work effectively because there is not any function that can be excluded. Chopper with the original setup still works on these cases because the excluded functions which were configured are called from the flaw-functions. However, it does not fit the purpose of patch testing, where modified functions (maybe become flaw-functions) are target of testing. Therefore, we think these cases are out of scope. For CVE 2015-2806, the result seems worse than the original Chopper, but it is still better than KLEE. It happens because the complexity threshold is not optimal. We iteratively tested threshold values from 2 to 10, and identified that on average, 5 was the most optimal setting to use. However, it must be noted that although this value is used by default, it is not the most suitable in every case. Finding the optimal threshold for every case is something we will pursue in our future work, as we believe it can be determined by the number of program paths. We experienced an error during the analysis for CVE 2014-3467(2), the exact reason for this is not clear, but we found that it happened because STP [23] can not solve the condition path. We presume that the reason is that Chopper missed a pointer when finding the side effects.

V. Conclusion

Chop's automatic exclusion is a novel exclude functions method for finding memory bugs for chopped symbolic execution with patch testing problems. Our preliminary evaluation shows that chop's automatic exclusion has better time running than the original Chopper in most cases. It proves that chop's automatic exclusion can be used in practice. By using our method, patch testing can be run automatically without prior expert knowledge of tester as using Chopper. However, our method still has some limitations, namely that an optimal complexity threshold cannot be ascertained automatically. Additionally, ChopperAEx can currently only work stably on C-based applications. In the future, we will try to address these limitations and develop ChopperAEx to support other languages. We will also seek to improve performance and find a more exact way to avoid excluding the functions that may affect the target function (e.g. indirect jump or jump via address).

References

[1] O. Crameri, N. Knezevic, D. Kostic, R. Bianchini, and W. Zwaenepoel, "Staged deployment in mirage, an integrated software upgrade testing and distribution system," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6, pp. 221–236, 2007.

[2] Z. Gu, E. T. Barr, D. J. Hamilton, and Z. Su, "Has the bug really been fixed?" in *2010 ACM/IEEE 32nd International Conference on Software Engineering*, vol. 1. IEEE, 2010, pp. 55–64.

[3] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram, "How do fixes become bugs?" in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 26–36.

[4] F. Jianming, C. Jing, P. Guojun *et al.*, "Pvdf: An automatic patch-based vulnerability description and fuzzing method," 2014.

[5] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

[6] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "Exe: Automatically generating inputs of death," *ACM Transactions on Information and System Security (TISSEC)*, vol. 12, no. 2, pp. 1–38, 2008.

[7] F. Wang and Y. Shoshitaishvili, "Angr-the next generation of binary analysis," in *2017 IEEE Cybersecurity Development (SecDev)*. IEEE, 2017, pp. 8–9.

[8] R. David, S. Bardin, T. D. Ta, L. Mounier, J. Feist, M.-L. Potet, and J.-Y. Marion, "Binsec/se: A dynamic symbolic execution toolkit for binary-level analysis," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 653–656.

[9] D. Babić, L. Martignoni, S. McCamant, and D. Song, "Statically-directed dynamic automated test generation," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, 2011, pp. 12–22.

[10] P. Godefroid, M. Y. Levin, D. A. Molnar *et al.*, "Automated whitebox fuzz testing." in *NDSS*, vol. 8, 2008, pp. 151–166.

[11] A. Chaudhuri and J. S. Foster, "Symbolic security analysis of ruby-on-rails web applications," in *Proceedings of the 17th ACM conference on Computer and communications security*, 2010, pp. 585–594.

[12] D. Trabish, A. Mattavelli, N. Rinetzky, and C. Cadar, "Chopped symbolic execution," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 350–360.

[13] P. D. Marinescu and C. Cadar, "Katch: High-coverage testing of software patches," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 235–245.

[14] D. Spinellis, "Git," *IEEE software*, vol. 29, no. 3, pp. 100–101, 2012.

[15] GitHub, "Github homepage," Accessed Sep. 17, 2021. [Online]. Available: https://github.com/

[16] GitLab, "Gitlab about page," Accessed Sep. 17, 2021. [Online]. Available: https://about.gitlab.com/

[17] SOURCEFORGE, "Sourceforge homepage," Accessed Sep. 17, 2021. [Online]. Available: https://sourceforge.net/

[18] Bitbucket, "Bitbucket homepage," Accessed Sep. 17, 2021. [Online]. Available: https://bitbucket.org/

[19] LLVM, "Llvm opt documents," Accessed Sep. 17, 2021. [Online]. Available: https://llvm.org/docs/CommandGuide/opt.html

[20] A. H. Watson, D. R. Wallace, and T. J. McCabe, *Structured testing: A testing methodology using the cyclomatic complexity metric*. US Department of Commerce, Technology Administration, National Institute of . . . , 1996, vol. 500, no. 235.

[21] LLVM, "Llvm 3.4 getting started," Accessed Sep. 17, 2021. [Online]. Available: https://releases.llvm.org/3.4.2/docs/GettingStarted.html

[22] M. Clemencic and P. Mato, "A cmake-based build and configuration framework," in *Journal of Physics: Conference Series*, vol. 396, no. 5. IOP Publishing, 2012, p. 052021.

[23] V. Ganesh, "Stp homepage," Accessed Sep. 17, 2021. [Online]. Available: https://stp.github.io/