Statically-Guided Fork-based Symbolic Execution for Vulnerability Detection

Yue Wang, Hao Sun, Qingkai Zeng
State Key Lab for Novel Software Technology, Nanjing University
Department of Computer Science and Technology, Nanjing University
Nanjing 210023, China
wxywang89@163.com, shqking@gmail.com, zqk@nju.edu.cn

***Abstract*—Fork-based symbolic execution would waste large amounts of computing time and resource on *invulnerable paths* when applied to *vulnerability detection*. In this paper, we propose a statically-guided fork-based symbolic execution technique for vulnerability detection to mitigate this problem. In static analysis, we collect all *valid jumps* along *vulnerable paths*, and define the priority for each program branch based on the ratio of vulnerable paths over total paths in its subsequent program. In fork-based symbolic execution, path exploration can be restricted to vulnerable paths, and code segments with higher proportion of vulnerable paths can be analyzed in advance by utilizing the result of static analysis. We implement a prototype named *SAF-SE* and evaluate it with ten benchmarks from GNU Coreutils version 6.11. Experimental results show that *SAF-SE* outperforms KLEE in the efficiency and accuracy of vulnerability detection.**

Keywords-fork-based symbolic execution; static analysis; vulnerability detection; program analysis

I. INTRODUCTION

Symbolic execution was first proposed by James C. King [1] in 1976. Fork-based symbolic execution uses symbolic values as inputs to execute target programs and replaces concrete program operations with ones that manipulate symbolic values during the execution. When program execution branches based on symbolic values, it follows each valid branch and collects the branch condition as the constraint of the corresponding path. When one path terminates or hits a bug, a test case will be generated by solving the collected constraints. Symbolic execution has two advantages: 1) having high code coverage and 2) producing no false positives.

Recently, fork-based symbolic execution has been applied to the field of *vulnerability detection*. The key challenge lies in that the goal of vulnerability detection is to expose vulnerable code as soon as possible and *vulnerable paths*, i.e. paths involving vulnerable code, only occupy a small proportion in programs, while fork-based symbolic execution selects branch blindly, leading to a considerable waste of computing time and resource on exploring *invulnerable paths*. Besides, the vulnerability detection accuracy would also be affected, i.e. generating false negatives, if computing time and resource is limited in real-world scenarios. Furthermore, *path explosion* would worsen both the efficiency and accuracy of vulnerability detection if target programs are in large scale.

To address this issue, we propose and implement a statically-guided fork-based symbolic execution technique for vulnerability detection, named *SAF-SE*. Static analysis process marks vulnerable paths and collects all valid jumps along them. These valid jumps would restrict symbolic execution to vulnerable paths only, and generate test cases which can violate the security constraints of sensitive operations. Furthermore, we define the priority for each program branch based on the ratio of vulnerable paths over total paths in its subsequent program. These branch scores are used by execution state selector to determine the priority of each execution state. Therefore, program segments with higher proportion of vulnerable paths will be explored first and more vulnerable code will be detected in the circumstance of limited computing time or resource. Hence, *SAF-SE* can not only accelerate fork-based symbolic execution process but also improve the accuracy of vulnerability detection.

This paper makes three contributions. First, we propose a statically-guided fork-based symbolic execution technique for vulnerability detection, in which we restrict fork-based symbolic execution on vulnerable paths. Second, we score program branches based on the ratio of vulnerable paths in subsequent program. Hence, code segments with higher proportion of vulnerable paths would be analyzed earlier. Third, we implement a prototype name *SAF-SE* and evaluate it with 10 benchmarks from GNU Coreutils 6.11. Experimental results show *SAF-SE* can improve vulnerability detection efficiency, and reduce false negatives when time and resource is limited.

II. DESIGN OF SAF-SE

Figure 1 illustrates the architecture of *SAF-SE*. It consists of three components: *graph generation* module, *static analysis* module and *fork-based symbolic execution* module. Note that users can define sensitive operations and corresponding security constraints in *user-defined configuration file*.

A. Graph Generation Module

Graph generation module reads LLVM bytecode file as input and generates the call graph and control flow graphs (CFGs). The call graph and CFG generation process in LLVM Utils doesn't consider dynamic link library functions. Therefore, we utilize a *light-weight symbolic executor* to obtain a relatively complete program. In it, we simulate the link process by executing the target program symbolically with the simplest

This work has been partly supported by National NSF of China under Grant No. 61170070, 61431008, 61321491; National Key Technology R&D Program of China under Grant No. 2012BAK26B01.
(DOI reference number: 10.18293/SEKE2015-094)

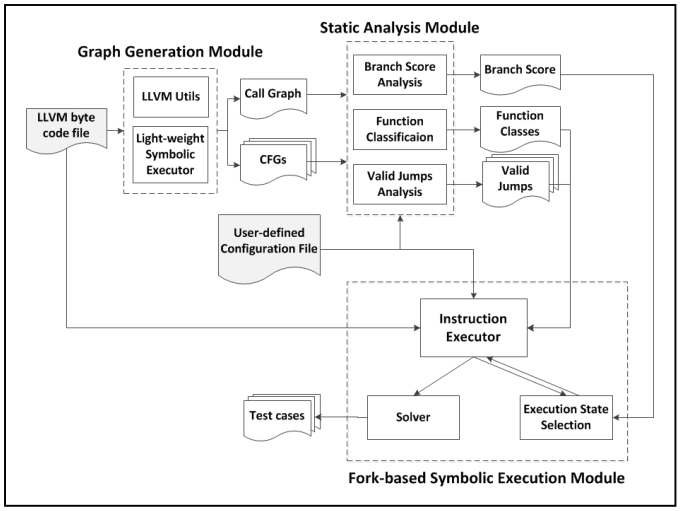

Figure 1. Architecture of SAF-SE

symbolic strategy, i.e. one symbolic input sized of one character in order to get dynamic library functions in records. Considering the small time consumption, i.e. less than 10 seconds, we call it *light-weight symbolic executor*.

B. Static Analysis Module

Static analysis module at first locates sensitive operations in target programs and divides functions into three categories. Then *valid jumps analysis* collects all valid jumps by marking the conditional values of each branch instruction that can lead to sensitive operations. At last, *branch score analysis* calculates the score of program branch according to the ratio of vulnerable paths over total paths in its subsequent program.

1) Function Classification

We define two attributes for each function, i.e. *vul-related* and *vul-lead*.

Definition 1. Function *f* is *vul-related*, if there are sensitive operations in *f*, or *f* calls another *vul-related* function.

Definition 2. If function *h* invokes function *f* at call site *loc*, and there are sensitive operations or *vul-related* function calls on the paths from *loc* to the exits of function *h*, then function *f* is *vul-lead*.

To calculate *vul-related* attribute, initially, we mark functions with sensitive operations inside as *vul-related*. Then, callers of *vul-related* functions are also marked as *vul-related*. To calculate *vul-related* attributes, first we locate the positions of sensitive operations and *vul-related* function call sites as *locs* in each function. Then, we initialize all the functions called between the function entry and *locs* as *vul-lead*. At last, all callees of *vul-lead* functions are also marked as *vul-lead*.

According to the attributes, functions in target programs can be divided into three categories, i.e. *T1~T3*, and different execution strategies would be applied to different categories.

a) T1: vul-lead = true: sensitive operations would be invoked after T1 functions. Since operations within T1 function might affect the sensitive operations afterward, all paths inside would be executed symbolically to gain conservative results.

Algorithm 1

input: CFG_{fn}, $V_{target\text{-}op}$
output: $Set_{valid\text{-}jmp}$
procedure calValidJmps (CFG_{fn}, $V_{target\text{-}ops}$)
1 $V_{sen\text{-}BBs}$ = collectSensitiveBBs (CFG_{fn}, $V_{target\text{-}op}$);
2 **for each** BasicBlock *bb* $\in V_{sen\text{-}BBs}$ **do**
3 $V_{parent\text{-}BBs}$ = collectParentBBs (CFG_{fn}, *bb*);
4 **for each** BasicBlock *pbb* in $V_{parent\text{-}BBs}$ **do**
5 Instruction *inst* = last branch instruction in *pbb*;
6 *validChoice* = getValidCondition (*pbb*, *bb*);
7 $Set_{valid\text{-}jmp}$.insert(<*fn*, *inst*, *validChoice*>);
8 $V_{sen\text{-}BBs}$.insert(*pbb*);
9 **return** $Set_{valid\text{-}jmp}$;

b) T2: vul-related = true and vul-lead = false: T2 functions have sensitive operations inside and have no sensitive operation after the execution of them. Hence, invulnerable paths inside can be pruned in symbolic execution.

c) T3: vul-related = false and vul-lead = false: T3 functions neither involve sensitive operations, nor have sensitive operations afterward. Hence, symbolic executor would terminate the execution process for T3 function calls.

2) Valid Jumps Analysis

Valid jumps analysis aims to collect the *valid jumps* of each branch instruction in T2 functions. *Valid jump* is the conditional value of branch instruction which can lead execution to sensitive operations. Algorithm 1 illustrates this process for function *fn*. $V_{target\text{-}op}$ refers to the set of sensitive operations and *vul-related* function calls. Tuple < *f*, *inst*, *choice*> is used to denote one valid jump, i.e. the instruction *inst* in function *f* would lead to sensitive operations under the conditional value *choice*. $Set_{valid\text{-}jmp}$ stores all the collected valid jumps.

Our valid jumps analysis is at basic block granularity. Initially, basic blocks involving $V_{target\text{-}op}$ are marked as sensitive (line 1). Then, for each sensitive basic block *bb*, we fetch each of its preceding basic block *pbb* (line 3) and set the branch from *pbb* to *bb* as a valid jump (lines 5 to 7). At last, *pbb* is also marked as sensitive (line 8). This process will continue until all sensitive basic blocks have been analyzed.

3) Branch Score Analysis

In branch score analysis, for each program branch, first we count the number of total paths and vulnerable paths, then score the branch based on the ratio of vulnerable paths over total paths in its subsequent program. These scores would be further used by execution state selector to explore the branch with higher proportion of vulnerable paths in advance. Note that we count the loop as one path when calculating the number of paths due to the lack of actual execution times of the loop structure in static analysis.

C. Fork-based Symbolic Execution Module

Fork-based symbolic execution module explores vulnerable paths following the branch scores and generates test cases which can violate security constraints for sensitive operations.

Generally speaking, it consists of three main parts: *instruction executor*, *execution state selector* and *constraint solver*.

1) Instruction Executor

We modify the instruction executor to analyze vulnerable paths with the results of function classification and valid jumps analysis. When we deal with sensitive operations, a verify process *check()* will be used to check if current constraints violate security constraints. If so, a test case would be generated by the constraint solver and reported to users.

When executing *Call* instructions, we check whether the callee is T3 function. If so, we would terminate current execution process and remove the execution state from the execution state pool based on the analysis in Section II-B-1).

As for *Branch* instructions and *Switch* instructions, we at first check whether current function belongs to T1 function. If so, we follow the original fork-based symbolic execution process. If current function is T2 function, execution flow can only be transferred to the valid succeeding basic blocks according to the result of valid jumps analysis. For each valid branch, we construct a new execution state by copying the current execution state, changing instruction pointer *pc* to the valid destination instruction, and adding condition expression into constraint set. At last, we insert the new execution states into the execution state pool.

2) Execution State Selector

Execution state selector aims to select an execution state from the execution state pool. Since existing selection strategies, e.g. depth-first search (DFS), breadth-first search (BFS), and covering new focus on program coverage, they cannot accelerate the process of vulnerability detection. Hence, we design a new selection strategy for vulnerability detection. Leveraging the results of branch score analysis, we select the execution state in the order of scores. In this way, code segments with high proportion of vulnerable paths would get analyzed in advance, accelerating vulnerable paths exploration and explore as many vulnerable paths as possible with limited computing time and resource.

III. Implementation and Evaluation

A. Implementation Details

We have implemented a prototype named *SAF-SE*. In it, we use a fork-based symbolic executor with one symbolic argument, whose size is one character, as the light-weight symbolic executor, and LLVM-3.1 utils to generate call graph and CFGs. As for the static analysis part, we implement a LLVM optimization pass written in about 1,600 lines of C++ on call graph and CFGs. In fork-based symbolic execution module, we adopt KLEE [2] and modify its instruction executor and the execution state selector based on previous discussion.

B. Experimental Setup

To evaluate the effectiveness of *SAF-SE*, we applied it on ten programs from GNU Coreutils version 6.11, and compared the results with KLEE [2]. In our experiments, we set seven library function calls as sensitive operations, including *alloc*, *malloc*, *realloc*, *calloc*, *memcpy*, *memccpy* and *memset*. All experiments were run on a machine with 3.20GHz Intel(R) Core(TM) i5-3470 processor and 4G of memory, running 64-bit Linux 3.2.0.

C. Results of Static Analysis

Table 1 shows the experimental results of static analysis on the ten benchmarks. The time cost (Column 2) in static analysis is negligible, averaging about 0.389s. Columns 3 to 5 show the distributions of three types of functions. We can observe that T2 functions, in which invulnerable paths can be pruned, account for the largest proportion, about 48.4% on average. Column 6 indicates the number of the basic blocks that can be pruned by static analysis, including all the basic blocks in T3 functions and those along invulnerable paths in T2 functions. On average, 21.8% of all basic blocks are free of symbolic execution.

D. Results of SAF-SE

To assess the effectiveness of *SAF-SE*, we look into the following two aspects: 1) we applied *SAF-SE* and KLEE on five benchmarks with the same arguments, respectively. For each benchmark, both *SAF-SE* and KLEE completed the whole symbolic execution process, and we assessed the reduction in execution time and executed instructions of *SAF-SE* over KLEE; 2) we applied *SAF-SE* and KLEE on the other five benchmarks with the same arguments, and we limited the execution time to 60 minutes, so as to assess the sensitive operation coverage promotion of *SAF-SE* over KLEE.

Table II shows the experimental results of the first aspect. Columns 2 to 4 show the results of KLEE, including the execution time, the number of analyzed instructions and the number of analyzed sensitive operations, and Columns 5 to 7 show those of *SAF-SE*. On average, *SAF-SE* achieved about 23.52% execution time reduction and about 23.17% analyzed instruction reduction over KLEE. In a word, *SAF-SE* can spend less execution time and execute fewer instructions than KLEE in completing the symbolic execution process without missing any sensitive operations.

Table III describes the experimental results of the second aspect. The meanings of the columns are similar to those in Table II. Each benchmark was run for 60 minutes with the same symbolic arguments: *--sym-args 1 5 10 --sym-files 2 100*, which means the number of symbolic arguments are from 1 to 5, and the length of each symbolic arguments is up to 10 characters. Meanwhile, we use two symbolic files which are not longer than 100 characters. From the results we can see that, *SAF-SE* executed 1.70x of instructions that KLEE executed, and discovered 1.37x of sensitive operations that KLEE covered in the same execution time. It is worth noting that the effectiveness in sensitive operation coverage promotion is highly dependent on the structure of each program and on the distribution of sensitive operations. We can conclude that *SAF-SE* can explore more vulnerable paths under limited execution time than KLEE.

The Experimental result proves *SAF-SE* can improve vulnerability detection efficiency of fork-based symbolic execution by pruning invulnerable paths in advance. Moreover, *SAF-SE* can reduce false negatives in the circumstance of limited computing time and resource with the help of branch scores from static analysis.

TABLE I. RESULTS OF STATIC ANALYSIS

program	time(s)	T1 (num/rate)	T2 (num/rate)	T3 (num/rate)	Prune BBs (num/rate)
mkdir	0.410	167/0.498	154/0.460	14/0.042	750/0.209
mkfifo	0.391	143/0.451	160/0.505	14/0.044	796/0.236
mknod	0.379	148/0.460	160/0.497	14/0.043	845/0.242
paste	0.363	146/0.458	159/0.498	14/0.044	777/0.226
ptx	0.685	196/0.513	167/0.437	19/0.050	886/0.158
seq	0.437	148/0.454	163/0.500	15/0.046	810/0.234
chmod	0.354	210/0.532	161/0.409	23/0.058	851/0.198
echo	0.281	132/0.423	165/0.529	15/0.048	841/0.251
basename	0.287	142/0.444	163/0.509	15/0.047	790/0.237
cat	0.300	148/0.460	159/0.494	15/0.046	816/0.234
AVG	0.389	158.0/0.470	161.1/0.484	15.8/0.046	816.2/0.218

TABLE II. RESULTS OF BENCHMARKS COMPLETED SYMBOLIC EXECUTION

program	KLEE			SAF-SE		
	time(s)	*instruction*	*sensitive op*	*time(s)*	*instruction*	*sensitive op*
echo	1548.01	40878183	20409	1045.72(-32.45%)	29702739(-27.34%)	20409
chmod	315.48	63130620	28425	243.52(-22.81%)	48300940(-23.49%)	28425
mkfifo	323.71	61459653	26275	228.82(-29.31%)	48493576(-21.10%)	26275
mknod	320.3	54853060	21937	259.67(-18.93%)	41718084(-23.95%)	21937
basename	25.45	5264958	2337	21.85(-14.16%)	4213381(-19.97%)	2337

TABLE III. RESULTS OF BENCMARKS RUN FOR 60 MINUTES

program	KLEE			SAF-SE		
	time(s)	*instruction*	*sensitive op*	*time(s)*	*instruction*	*sensitive op*
paste	3832.64	47913926	29752	3644.73	163029964(+240.26%)	61813(+107.76)
ptx	3706.1	99826914	40368	3718.47	102729476(+2.91%)	41703(+3.31%)
cat	4169.63	11818118	16565	4179.68	11868664(+0.43%)	16921(+2.15%)
seq	3669.66	81698491	35033	3668.82	103876566(+27.15%)	40830(+16.55%)
mkdir	3699.58	315383144	50299	3697.19	574760703(+82.24%)	76906(+52.90%)

IV. RELATED WORK

KLEE [2] is a widely used fork-based symbolic execution tool evolves from EXE [3]. Vitaly Chipounov et al. [4] proposed selective symbolic execution and implemented S2E by adopting KLEE as symbolic executor and using QEMU [5] to simulate execution environment. Combining symbolic execution with concrete execution, Patrice Godefroid et al. proposed the first concolic symbolic execution tool SAGE [6] for binary code. Symbolic execution has been widely used in vulnerability detection. SmartFuzz [7] leverages concolic symbolic execution to find integer bugs in x86 binary programs. Crashmaker [8] optimized the generational search algorithm in SAGE. However, these techniques still have to traversal the whole program even when detecting specific sensitive operations, while *SAF-SE* can improve the efficiency and accuracy of vulnerability detection by restricting path exploration on vulnerable paths.

V. CONCLUSION

In this paper, we propose a statically-guided fork-based symbolic execution technique for vulnerability detection and developed a prototype *SAF-SE* to restrict path exploration on vulnerable paths and to explore code segments with higher proportion of vulnerable paths earlier by utilizing the results of static analysis. We evaluated *SAF-SE* with ten benchmarks from GNU Coreutils version 6.11, and compared it with KLEE. The experimental results show that, *SAF-SE* improves the efficiency of vulnerability detection a lot, and reduces generating false negatives in the circumstances of limited computing time and resource.

REFERENCES

[1] J.C..King, "Symbolic execution and program testing", ;inproceedings of Communications of the ACM, 1976, pp.385-394

[2] C.Cadar, D.Dunbar, and D.Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs", ;inproceedings of OSDI'08 Proceedings of the 8th USENIX conference on Operating systems design and implementation, 2008, pp.209-224

[3] C.Cadar, V.Ganesh, P.M..Pawlowski, D.L..Dill, and D.R..Engler, "EXE: automatically generating inputs of death", ;conference of Computer and Communications Security, 2006, pp.322-335

[4] V.Chipounov, V.Kuznetsov, and G.Candea, "S2E: a platform for in-vivo multi-path analysis of software systems", ;inproceedings of Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems, 2011, pp.265-278

[5] F.Bellard, "QEMU, a fast and portable dynamic translator", ;inproceedings of ATEC '05 Proceedings of the annual conference on USENIX Annual Technical Conference, 2005, pp.41-41

[6] P.Godefroid, M.Y..Levin, and D.A..Molnar, "Automated Whitebox Fuzz Testing", ;conference of Network and Distributed System Security Symposium, 2008, pp.-1—1

[7] D.Molnar, X.Cong.Li, and D.A..Wagner, "Dynamic test generation to find integer bugs in x86 binary linux programs", ;inproceedings of SSYM'09 Proceedings of the 18th conference on USENIX security symposium, 2009, pp.67-82

[8] Bing Chen, Qingkai Zeng, and Weiguang Wang. "Crashmaker: an improved binary concolic testing tool for vulnerability detection." inproceedings of the 29th Annual ACM Symposium on Applied Computing. ACM, 2014, pp.1257-1263