FCEP: A Fast Concolic Execution for Reaching Software Patches

Meng Fan[1,2], *Wenzhi Wang[1,2], Aimin Yu[1], Dan Meng[1]
[1]*Institute of Information Engineering, Chinese Academy of Sciences*
[2]*School of Cyber Security, University of Chinese Academy of Sciences*
Beijing, China
{fanmeng,wangwenzhi,yuaimin,mengdan}@iie.ac.cn
* Corresponding Author: wangwenzhi@iie.ac.cn

***Abstract*—Software updates that bring new features to the users or that fix old errors can easily introduce new errors, which makes it necessary for users to repeatedly consider whether to update the software to the latest version. Therefore, the security testing for updated software is indispensable before its releasing. State-of-the-art increasing number of researchers have been devoting to develop new techniques that can automatically generate high-coverage test suites and detect software errors introduced by patches. In this paper, we proposed a technique based on concolic execution to ensure the correctness and reliability of a patch. Our method generates test inputs to cover the changed lines of the patch and the relevant function by using a target-based search strategy which combines the selector based on the mapped address and the selector based on the priority. A prototype system called FCEP was implemented and evaluated with 5 C-programs. The experimental results demonstrated that our method reaches the new code introduced by patches quickly and achieves a high coverage.**

***Index Terms*—concolic execution, patch testing, Search strategies**

I. Introduction

The extensibility of software is both a blessing and a curse. On the one hand, one can easily add new functionality or patches to fix incorrect behavior. On the other hand, any software changes may introduce unexpected errors and security vulnerabilities, which are disasters for users and make users hesitate to update their software. As matter of fact, many people prefer to not upgrade their software to the latest version [1], [2], but rely on earlier versions which also usually contain serious errors and reduced functionality. Therefore, it is very necessary to perform a comprehensive test of the updated software. However, software testing is expensive and time-consuming as it involves writing a large numbers of manual test suites to validate various paths. This is a tedious process that requires an immense amount of work and a good understanding of the tested system. Some recent testing effort focuses on code that has changed from one version to the next [3]–[7].

Patches, as a typical form of software changes, ideally, should be comprehensively tested, but this level of testing is still far from being achieved in practice [8]. State-of-the-art some studies [6], [9], [10] test the patches based on concolic execution [11], [12], which is a program analysis technique that provides the ability to generate inputs to form high-coverage test suites. Concolic execution has proved to be a good choice to comprehensively test real software [13], [14], with its ability to systematically explore different program paths. Most of the work on concolic execution is focused on whole program tests, in which all parts of the program are treated equally. However, the number of execution paths in a program is usually exponential in the number of branches, exploring all possible execution paths is infeasible. That is, concolic execution faces challenges of path explosions [11], [15]. An efficient search strategy of concolic execution is very important to overcome the path explosions challenge in testing patches. Despite recent progress, the studies still far from reaching the goal of fast and automatically generating test cases that contain code changes in the actual program.

In this paper, we have developed an automated testing method called FCEP based on concolic execution to ensure the correctness and reliability of patches, which takes advantages of both static and dynamic analysis to generate test inputs to cover the changed lines in patches. Instead of exploring all branches in the candidate list, FCEP searches priorities branches according to the distance between the uncovered statements and the targets (e.g. lines in patch). FCEP combines the exploration of the patch with the exploration of the function where the patch is located and dynamically adjusts the search target to conduct a more comprehensive test of the patch-related functions.

This paper mainly makes following contributions:

- FCEP ensures the reliability and security of software updates by using a target-based search strategy to test the patch and the relevance function quickly, which combines the selector based on the mapped address and the selector based on the priority, to test the patch and its relevant function quickly.
- The selector based on the mapped address reaches the patch quickly by calculating the shortest distance between candidate states and targets (e.g. lines in patch). The selector based on the priority comprehensively tests the relevant function of the patch as soon as possible by assigning priority to the related states.
- FCEP reduces the false negative by comprehensively

DOI reference number: 10.18293/SEKE2021-018.

testing the relevant function of patches and modifying the CFG (Control Flow Graph) in real-time based on the results of concolic execution. For example, guiding the path search in dynamic analysis only according to the results of static analysis cannot cover relevant paths containing indirect jumps.

- We performed experiments on 5 C-programs. FCEP covered more than 90% of the patch lines and found 34 out of 39 bugs in 5 tested software in the least amount of time. The experiments showed that FCEP can cover more lines of code in patches, and can quickly find bugs introduced by patches.

The rest of the paper is organized as follows: Section II introduces concolic execution and describes several representative search strategies found in the literature; Section III details our approach; Section IV shows the evaluation plan and the experimental results; Section V discusses related work, and Section VI concludes.

II. Background

a) Concolic Execution: Concolic execution is an automatic test generation techniques based on symbolic execution, a program analysis technique that can systematically explore paths through a program. The key idea behind symbolic execution is to run the program with symbolic values instead of concrete ones. Then, whenever an encountered branch is directly or indirectly dependent on the symbolic input, execution determines the feasibility of both sides of the branch, and creates two new independent symbolic states which are added to a worklist to follow each feasible side separately. Finally, whenever a path terminates or hits an error, the constraints on that path are solved to produce a concrete input that exercises the path. Since the number of execution paths in a program is usually exponential in the number of branches, exploring all possible execution paths is infeasible. To address this problem, concolic execution relies on the search heuristic to steers concolic execution in a way to maximize code coverage in a given limited time budget [11].

b) Search strategies: Since enumerating all paths of a program can be very expensive, in many software engineering projects related to testing and debugging, the search is prioritized by looking at the most promising paths first. Depth-first search (DFS) and breadth-first search (BFS) are the most common strategies. DFS expands a path as much as possible before backtracking to the deepest unexplored branch, while BFS expands all paths in parallel. DFS is often adopted when memory usage is at a premium. The breadth-first search (BFS) strategy traverses the execution tree according to a BFS order. The BFS strategy prefers branches that appear early in the execution paths, therefore generating new input vectors is easier because a smaller number of constraints will be involved for those branches. Hence, in spite of the higher memory pressure and of the long time required to complete the exploration of specific paths, some tools resort to BFS. In theory, both DFS and BFS strategies can cover all execution paths in the execution tree. However, as described in the previous section, real world programs have a nontrivial number of execution paths and neither strategy scales to even medium-sized programs [11], [12], [16]. Another popular strategy is random path selection [14], which has been refined in several variants.

III. System Design and Implimentation

In this section, we first show an overview of FCEP. We then explain the address mapping technology an the strategy of path selection.

A. Overview

Fig. 1 demonstrates an overview of FCEP. The inputs of our technique are: 1) the new version of program and the patch, and 2) the inputs selected from the test suite. The output is a set of inputs that trigger crash bugs or cover the code of patch.

On the one hand, FCEP uses the disassembler to generate the CFG and the CG (Call Graph) of the new version program. FCEP marks the position in the CFG for each target which is a line of the patch, and generates a list of function address ranges based on the CG. On the other hand, the executor creates a new state when it encounters a branch. And at the same time, the address finder searches the address of the next instruction in the generated CFG for both states. After mapping the new state to the address of the next instruction, FCEP puts the newly generated state into the candidate pool to wait for the next time selecting of the selector to complete concolic execution.

When the path entered with the initial value has executed, FCEP selects a new execution state by adopting a target-based search strategy in the candidate pool to perform concolic execution. Concolic exploration that focuses on target-based search provides inputs for a crashing path. Our search strategy infers the paths which are not covered by patches to avoid exploring large numbers of paths, and to direct the search towards the paths covered by patch.

Once there is a state that triggers the function where the patch is located, the target-based search strategy chooses the new state which falls into the state of the target function as much as possible. This is mainly because for the testing of patch, we believe that only covering the line of patch is far from meeting the testing requirement of ensuring the security of the patch. At the same time, FCEP uses the concolic execution to correct the paths which through indirect jumps or the function pointer calls in the statically generated CFG. Once indirect jumps or function pointer calls are encountered in the tested paths, FCEP will splice the related indirect jump blocks to find more paths.

B. Address Mapping

a) Generate CFG&CG: The first step of our analysis is determining the differences between the new program version and its previous version, (i.e. the patch). Theoretically, each line in the patch is a potential target to our FCEP. Whereas, many lines can be overlooked in practice because the source code contains many non-executable lines (e.g., declarations,

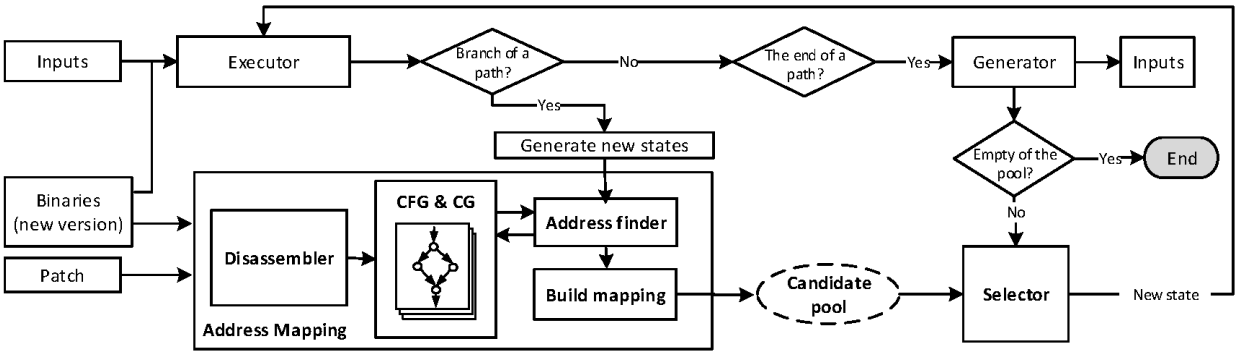

Fig. 1. A high-level overview of our execution.

comments, blank lines, or lines not compiled into the executable). The patch is divided into several sets according to the functions they belong to, and each set is processed separately in the subsequent steps. FCEP selects a line of the patch in each set as a core target, and this line represents the core modification of the function as much as possible. FCEP statically analyzed the new version program using IDA Pro toolset to establish a control-flow graph (CFG) and a call graph (CG). FCEP uses a core target to mark possible execution paths in CFG, and marks the corresponding function call paths in CG. Note that considering that indirect jumps and pointers may cause inaccuracy in static analysis, FCEP dynamically corrects the CFG and CG in real-time in subsequent steps.

b) Build mapping: Concolic Executor uses the initial input to test the new version program. When the path encounters a branch, executor generates a new path state and notifies the address finder to find the address of the next instruction for the new state in the statically generated CFG. FCEP maps the new state to this instruction address and puts the new state into the candidate pool so that it can be used in subsequent path selection. The program to be tested must be translated into intermediate language to interpret and execute. When a branch is encountered during the concolic execution, it adds the newly generated branch state to the execution tree. FCEP constructs the execution tree following the same method in static analysis, putting the true branches in the branch state in the left sub-tree and the false branches in the right sub-tree. As the true branches and the false branches of the state have been adjusted to be consistent with the CFG generated in static analysis while creating the new state, the address of next instruction corresponding to each branch can be queried quickly.

c) Dynamically modify CFG&CG: What we need to pay attention to is that because the static analysis cannot accurately infer indirect jumps or the function pointer calls, there is a false negative when using static analysis methods to guide the concolic execution. If FCEP encounters indirect jumps or the function pointer calls during concolic execution, it dynamically modifies the CFG and CG to obtain more accurate information of path.

C. The Target-based Search Strategy

a) Selector based on the mapped address: New states are generated where the conditional branch is located by executor, and FCEP put them into a candidate pool for subsequent selecting. The selector selects a new state from the candidate pool following a search strategy to continue the concolic execution when the execution of a path is finished. Since the search strategy usually affects the coverage of concolic execution, careful selection of the algorithm can help to achieve the desired goal.

Before the start of the test, FCEP has obtained the address range of each function from the CG, and has selected a path marked in the CFG which can execution from the function main to the selected core target. When the execution of the initial state is finished, FCEP determines whether the state newly generated hits the target function by the address of the next instruction, in other words, whether the address is within the range of the target function. If it hits, FCEP selects the path closest to the core target to continue running. If there is no state hits the target function, FCEP looks for the caller of the target function and confirm whether there is a state hits the caller. By analogy, FCEP selects the closest one among states which hit the caller to continue execution. Since it has been confirmed whether the new state previously generated hits the target function or its callers, FCEP gives priority to the new states derived from the current running state to confirm whether they fall into the target function.

As we all know, code addresses of a program are not completely continuous, but they are continuous in a same function of a program. Therefore, when selecting the closest state, FCEP calculates distance between the new state and the target using the formula $Di = |statei\_addr - target\_addr|$. For the target function, the target is a line of patches determined before running the program, and for other functions in the call chain of the target function, the target is the line of the function call.

b) Selector based on the priority: When a new branch state is generated, FCEP determines whether its next instruction address hits the target function (that is, the function where the patch is located). If it hits, FCEP gives priority to the state. When a path hits the core target code (that is, one line of the

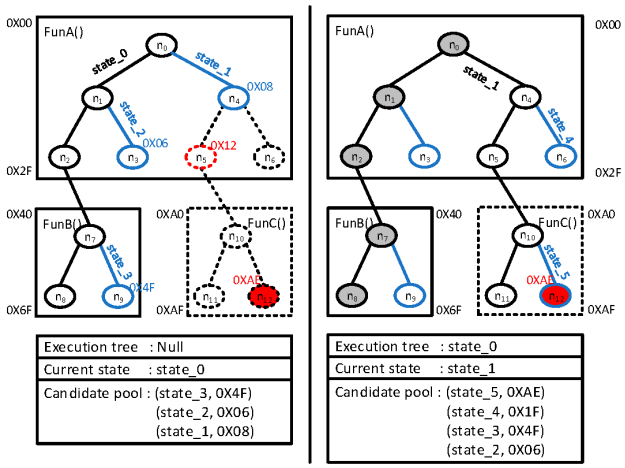

Fig. 2. An example of our search strategy.

patch), FCEP continues to select the state that hits the target function to perform the concolic execution until there is no state hits the target function or the expected time runs out. In this step, FCEP only needs to execute the state with priority until all the states with priority have been executed.

After that, it continues to perform all the above steps for the next core target code. The purpose of running the states that hit target function is to test as much code near the patch as possible. Because for the security testing of patches, only paying attention to the path covered by the patch code itself is often unable to meet the requirements. The code near the patch is often associated with the patch, so it is necessary to test the function where the patch is located.

D. An Example of Search Strategy

Fig. 2 is an example of the search strategy, and it is a combination of CFG and CG. Assume that the address range of each function obtained from CG is $FunA(0x00, 0x2F)$, $FunB(0x40, 0x6F)$ and $FunC(0xA0, 0xAF)$. The core target selected in the target function $FunC$ is located at node n_{12}, and a path sequence selected in the function call chain passing through the core target is $\{n_0,\ n_4,\ n_5,\ n_{10},\ n_{12}\}$. FCEP determines the sequence of the function call node and its address as $\{n_5\ (0x12)\}$. The execution tree is empty when the concolic execution starts. Suppose $state\_0$ is the initial state and it corresponds to the sequence of $\{n_0,\ n_1,\ n_2,\ n_7,\ n_8\}$ when FCEP runs the program with the initial inputs. At the same time, this state creates three new branch states $\{state\_1,\ state\_2,\ state\_3\}$ in the concolic execution. FCEP queries the address of the next instruction for the three new states in the CFG (that is, the address of the next block in the CFG) and puts them into the candidate pool. Assuming that the address of the next instruction of $state\_1$ is $0x80$, the address of $state\_2$ is $0x06$, and the address of $state\_3$ is $0x4F$.

When the state $state_0$ is executed, the selector selects a new state from the candidate pool. FCEP first determines whether the addresses of the next instruction in the three newly generated states hit the target function FunC. If there are some states in the range of the target function, FCEP selects a state closer to the core target node to continue the concolic execution. If there is no such state, the address funder queries whether there are states in the range of the caller function(e.g. the function $FuncA$). As shown in the Fig. 2, the three states do not hit the target function $FunC$, but there are two states $\{state\_1,\ state\_2\}$ hit the upper function of $FunC$, that is, $FuncA$. The node $n_5(0x12)$ is the point that the $FuncA$ calls the $FunC$. FCEP calculates all the distances between the states and the address of node n_5, that is, $D_1 = 4$, $D_2 = 6$, so it chooses the $state\_1$ which has a smaller distance as the next execution state.

Assuming the execution path of the $state_1$ is $\{n0,\ n4,\ n5,\ n10,\ n11\}$. It generates two new states $\{state\_4,\ state\_5\}$, and their addresses are $0x1F$ and $0xAE$. FCEP gives priority to judge whether the newly generated states hit the target function and finds the $state\_5$ has located in the target function, so this state is selected as the next execution state. After that, FCEP continues to select the state derived from $state\_5$ in the target function to complete the concolic execution.

E. Implementation

We implemented the proposed FCEP as a plugin of S2E [17] which is a general concolic execution framework. This plugin is mainly composed of three custom modules:

a) Automatic constructing CFG&CG: This module builds CFG and CG for the tested program and marks the selected target and path.

b) Address finder and mapping: This module finds the address of the next instruction in the CFG for the new state when the executor creates a branch state, and puts the mapped address into the candidate pool. When indirect jumps or function pointer calls is executed, this module dynamically corrects CFG and CG.

c) The target-based selector: This module uses the target-based search strategy to select a new state to continue concolic execution after a path is finished. This module combines the exploration of the patch with the exploration of the function where the patch is located to dynamically adjust the search target.

IV. Evaluation

We evaluated FCEP experimentally with real-world application binaries, answering the following research question:

- Effectiveness of generated heuristics: Can FCEP generate effective search heuristics? What is the coverage for patches?
- Bug detecting ability: Does FCEP generate effective search heuristics and how faster FCEP detect target bugs than the current concolic execution techniques?

We conducted all of the experiments on a computer running Ubuntu 18.04 64-bit, equipped with a 3.4 GHz Intel Core i7-6700 CPU and 24 GB of RAM. We evaluated FCEP with software patches from GNU Coreutils application suite. We only tested 8 programs that contain errors in the Coreutils

TABLE I
INFORMATION OF PATCHES AND BUGS

Targets	Lines	Func.	Target bugs	Patches (LoC)	Patches (Func)
Coreutils-6.10	4570	93	8	34	8
Grep-2.0	5956	132	6	53	6
Make-3.75	28715	555	10	109	10
Sed-1.17	4085	73	3	71	2
Vim-5.0	66209	1749	12	262	16
Sum	109535	2602	39	529	42
Average	21907	520	7.8	105.8	8.4

TABLE II
LINE COVERAGE FOR PATCHES OF EACH TARGET

Targets	LoC of Patch related func.	S2E	KATCH	FCEP
		Line cov.	Line cov.	Line cov.
Coreutils	234	44.80%	69.71%	88.90%
Grep	353	41.61%	58.15%	94.67%
Make	909	53.27%	63.49%	89.07%
Sed	171	29.89%	55.36%	87.63%
Vim	1262	48.52%	70.04%	91.03%
Average	505.8	43.62%	63.35%	90.26%

test set, including: paste, pr, tac, mkdir, mkfifo, mknod, ptx and seq. Furthermore, we collected real-world bugs (shown in TABLE I) from SIR [18] C programs which were fixed by the original developers from Dec 1996 to July 2018. TABLE I shows the detail of the 8 tools in Coreutils and the 4 software (Grep, Make, Sed and Vim).

A. Effectiveness

In order to determine the effectiveness of heuristics, we ran S2E, KATCH [6] and FCEP with the above test software and patches for 100 hours respectively. The results are displayed in TABLE II. The second column of the table is the total number of lines of the function where the patches are located. FCEP achieved an average line coverage of 90.26%, which is 1.42 (90.26/63.35) times larger than that of KATCH and 2.07 (90.26/43.62) times larger than that of S2E. Because FCEP not only tests the line of the patch itself, it also tests other codes in the function where the patches are located, so its coverage is much higher than other testing tools. Experiment shows that in the same time, FCEP can concentrate resources on comprehensive testing where the patches are located.

As a matter of fact, some patches are macro-defined code blocks which were not compiled in our compiled environment, so that some patches were not covered. Dynamic symbol execution for a macro-defined code blocks is a common problem for they may not be compiled. Some patches are referenced header files and newly defined variables. For newly defined variables, the location where the variable is referenced can be tested. New variable definitions and new header files do not cause problems because they are reflected in the code which is really changed and are tested by that code. In our experiments, the macro-defined code described above is excluded whereas the remained patches were covered.

TABLE III
TARGET BUGS DETECTED BY AND THE EXECUTION TIME

Targets	Bugs	T(h)	S2E	KATCH	FCEP*	FCEP
Coreutils	8	94	3	6	7	8
Grep	6	92	2	3	5	5
Make	10	126	2	4	5	8
Sed	3	62	2	2	3	3
Vim	12	202	3	6	9	10
Sum	39	576	12	21	29	34
Average	#	#	48	27.43	19.86	16.94

B. Bug Detecting Ability

Coreutils-6.10 contains 8 vulnerabilities (paste, pr, tac, mkdir, mkfifo, mknod, ptx and seq) and there are 6, 10, 3, and 12 bugs in Grep, Make, Sed, and Vim, respectively. TABLE III summarize the number of detected bugs and the time spending for the three methods. The third column in the table is the running time of each program. In particular, the sixth column of the table (FCEP*) lists the data obtained when the target patches is covered, and does not include the data of a comprehensive search for the function where the patch is located.

For a same software, FCEP successfully detected more bugs than the other two. KATCH found 21, S2E found only 12, whereas FCEP detected 34 over 39 bugs in total, showing much higher bug detect rate for patches. Notably, some bugs are not crash errors so that platform is difficult to detect these bugs without adding assertions to the code. So FCEP still missed 5 errors. FCEP found 34 bugs in 576 hours, while KATCH took 576 hours to find 21 bugs and S2E found 12 bugs. The time period required for FCEP to detect a bug is 16.94 (576/43) on average, whereas those for KATCH and S2E are 27.43 (576/21) and 48 (576/12) respectively. FCEP*, which only tests the line where the patch is located and does not fully test the corresponding function, the time for finding each bug is 19.86(576/29).

Compared with S2E, FCEP finds more bugs in the same time. This is because it can reach the code block where the patch is located more quickly and concentrate resources to test the location of the target to reduce the exploration of redundant paths. Compared with KATCH, FCEP can correct the CFG obtained by static analysis in real time to detect the corresponding paths which through indirect jumps or the function pointer calls. Compared with FCEP*, which only detects the patch code line, FCEP can find more bugs in the same time because it can perform a more comprehensive test on the function where the patch is located.

V. RELATED WORK

In recent years, there has been a lot of research on bug search in programs based on patches, but the technical methods used are also different. SPAIN [19] is a patch analysis framework to automatically learn the security patch patterns and vulnerability patterns, and identify them from the program binary executables. But SPAIN focus on patches in which only

one function is modified for one patch, but do not support patches where multiple functions are changed for one patch.

Based on derived operation semantic and constraint formula from patched differences, PVDF [20] computes the semantic of patches for privilege elevation vulnerabilities. This work is similar to SPAIN, but it assumes the availability of patches, and only focuses on one particular vulnerability type. Differently, SPAIN attempts to summarize patterns for different vulnerability types, and only requires the binary programs but not the patches.

Shadow symbolic execution [10] is a novel technique for generating inputs that trigger the new behaviors introduced by software patches. However, Shadow is not fully automatic, while many of the annotations added could be automated, manual assistance might still be needed.

Several heuristic-based approaches have been proposed to guide an execution toward a specific branch. KATCH [6] is a technique for patch testing that combines symbolic execution with several novel heuristics based on program analysis that effectively exploit the program structure and existing program inputs. Compared with manual testing, despite the increase in coverage and the bugs found, KATCH was still unable to cover most of the targets. Because it does not handle with the paths which through indirect jumps or function pointer calls.

VI. Conclusion

Software updates are easy to introduce bugs, so a full test of the software patch is indispensable, but extremely expensive and time costing. In this paper, we develop a method called FCEP to ensure the reliability and security of software updates by using a target-based search strategy to test the patch and the relevance function quickly, which search strategy combines the selector based on the mapped address and the selector based on the priority. In addition, FCEP reduces the false negative by comprehensively testing the relevant function of the patch and modifying the CFG in real-time based on the results of concolic execution.

Experiments initially show that FCEP can lead to significant improvements in reducing the number of path to explore and the time-cost to reach the patch-related code. So it can exclude uninteresting parts of code during analysis and focuses on those paths most relevant to the patches. At present, FCEP can only solve part of the problems of indirect jumps and function pointer calls. In the future, we will further study the automatic identification of them.

Acknowledgment

This work is supported by the strategic Priority Research Program of Chinese Academy of Sciences, Grant No.XDC02010400.

References

[1] Zhongxian Gu, Earl T Barr, David J Hamilton, and Zhendong Su. 2010. “Has the bug really been fixed?” In 2010 ACM/IEEE 32nd International Conference on Software Engineering, Vol. 1. IEEE, 55–64.

[2] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. 2011. “How do fixes become bugs?” In Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering. ACM, 26–36.

[3] Domagoj Babi´c, Lorenzo Martignoni, Stephen McCamant, and Dawn Song. 2011. “Statically-directed dynamic automated test generation,” In Proceedings of the 2011, International Symposium on Software Testing and Analysis. ACM, 12–22.

[4] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S Foster, and Michael Hicks. 2011. “Directed symbolic execution,” In International Static Analysis Symposium. Springer, 95–111.

[5] Suzette Person, Guowei Yang, Neha Rungta, and Sarfraz Khurshid. 2011. “Directed incremental symbolic execution,” In Acm Sigplan Notices, Vol. 46. ACM, 504–515.

[6] Paul Dan Marinescu and Cristian Cadar. 2013. “KATCH: high-coverage testing of software patches,” In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. ACM, 235–245.

[7] Kunal Taneja, Tao Xie, Nikolai Tillmann, and Jonathan De Halleux. 2011. “eXpress: guided path exploration for efficient regression test generation,” In Proceedings of the 2011 International Symposium on Software Testing and Analysis. ACM, 1–11.

[8] Paul Marinescu, Petr Hosek, and Cristian Cadar. 2014. “Covrig: A framework for the analysis of code, test, and coverage evolution in real software,” In Proceedings of the 2014 International Symposium on Software Testing and Analysis. ACM, 93–104.

[9] David A. Ramos and Dawson R. Engler. “Under-constrained symbolic execution: Correctness checking for real code.” In Proceedings of the 24th USENIX Conference on Security Symposium (SEC’15). USENIX Association, pp. 49–64, 2015.

[10] Tomasz Kuchta, Hristina Palikareva, and Cristian Cadar. 2018. “Shadow symbolic execution for testing software patches,” ACM Transactions on Software Engineering and Methodology (TOSEM) 27, 3 (2018), 10.

[11] Cristian Cadar and Koushik Sen. 2013. “Symbolic execution for software testing: three decades later,” Commun. ACM 56, 2 (2013), 82–90.

[12] Ting Chen, Xiao-song Zhang, Shi-ze Guo, Hong-yuan Li, and YueWu. 2013. “State of the art: Dynamic symbolic execution for automated test generation,” Future Generation Computer Systems 29, 7 (2013), 1758–1773.

[13] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. 2011. “Parallel symbolic execution for automated real-world software testing,” In Proceedings of the sixth conference on Computer systems. ACM, 183–198.

[14] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs,” In OSDI, Vol. 8. 209–224.

[15] Roberto Baldoni, Emilio Coppa, Daniele Cono D’Elia, Camil Demetrescu, and Irene Finocchi. A Survey of Symbolic Execution Techniques. ACM Comput. Surv. 51, 3, Article 50 (May 2018), pp.50:1 39, 2018

[16] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil Mcminn. 2013. “An orchestrated survey of methodologies for automated software test case generation,” J. Syst. Softw. 86, 8 (August 2013), 1978–2001.

[17] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. “S2E: A platform for in-vivo multi-path analysis of software systems.” In ACM SIGARCH Computer Architecture News, Vol. 39. ACM, 265–278.

[18] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. 2005. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and Its Potential Impact. Empirical Software Engineering 10, 4 (Oct. 2005), 405–435.

[19] Zhengzi Xu, Bihuan Chen, Mahinthan Chandramohan, Yang Liu, and Fu Song.2017. “SPAIN: security patch analysis for binaries towards understanding the pain and pills,” In Proceedings of the 39th International Conference on Software Engineering. IEEE Press, 462–472.

[20] S. Letian, Fu Jianming, Chen Jing and Peng Guojun, ”PVDF: An automatic Patch-based Vulnerability Description and Fuzzing method,” 2014 Communications Security Conference (CSC 2014), Beijing, 2014, pp. 1-8.