Analysis of Program Patches Nature and Searching for Unpatched Code Fragments *

Mariam Arutunian
Ivannikov Institute for System Programming of the Russian Academy of Sciences, Russian-Armenian University
Yerevan, Armenia
arutunian@ispras.ru

Hayk Aslanyan
Ivannikov Institute for System Programming of the Russian Academy of Sciences, Russian-Armenian University
Yerevan, Armenia
hayk@ispras.ru

Vahagn Vardanyan
Ivannikov Institute for System Programming of the Russian Academy of Sciences
Yerevan, Armenia
vaag@ispras.ru

Vahagn Sirunyan
Ivannikov Institute for System Programming of the Russian Academy of Sciences, Branch of Lomonosov Moscow State University in Yerevan
Yerevan, Armenia
sirunyan@ispras.ru

Shamil Kurmangaleev
Ivannikov Institute for System Programming of the Russian Academy of Sciences
Moscow, Russia
kursh@ispras.ru

Sergey Gaissaryan
Ivannikov Institute for System Programming of the Russian Academy of Sciences
Moscow, Russia
ssg@ispras.ru

***Abstract*—Software developers often copy and paste code within a project. Due to the possible existence of defects in the initial code fragment, this can lead to defects propagation across the project. Software changes in new version (patches) usually contain bug fixes, which can be used for detecting similar defects in a project. The purpose of this work is to develop method for analyzing the nature of patches between versions of executables and finding unpatched code fragments. At first, two versions of executables are compared for finding common and changed parts of code. Then, the method determines patches that can possibly be fixes of bugs. The final step is detection of unpatched code fragments. It is based on finding all clones of the buggy code fragments found in previous step which are not patched in the new version of the program. These fragments possibly contain defects. Developed tool allows to analyze programs of several architectures (x86, x86-64, arm, mips, powerpc). The experimental results show that the average percentage of true positive rate on the CoreBench test suite is 73%.**

Keywords— code static analysis; patch analysis; code clones; binary code analysis

I. Introduction

Programs patches are usually supposed to fix bugs. There are several methods [1] [2] [3] [4] [5], which help analytics to analyze patches, however they also need manual work. The method for analysis of patches nature described in this article is fully automatic.

Additionally, the proposed method finds unpatched fragments. Programmers, while writing a program, often use ready code fragments that may contain defects [6]. Then these defects can be duplicated elsewhere in the code by full or partial copy of the fragment with a defect. Subsequent correction of defects, for various reasons, does not guarantee correction in all clones of the fragment. The detection of such defects can be performed by analyzing patches between versions of program and searching for code clones for buggy fragment, which are not patched in new version.

Most of the existing tools for analyzing program patches are on the basis of the source code [7] [8]. In case of complete or partial absence of the project's source code, these methods cannot be applied. The method suggested in this article overcomes this limitation. The tool receives two versions of executable files, disassembles them, converts to platform independent intermediate representation, generates program dependence graphs (PDG) [9] and call graphs (CG). Then it compares executables using this information and detects common and changed code fragments based on previously developed method [1]. After detecting changed code fragments, method only considers patches, which can be defects fixes (the fragments in old version are considered as buggy). The final step is detection of unpatched code fragments. For this purpose, the clones of buggy fragments are found, which are not patched in new version. Thus, they possibly contain defects. As algorithm is based on platform independent intermediate representation, it allows to use the tool for several architectures (for which translation from assembler to the intermediate representation is supported).

The rest of the paper is organized as follows. In section 2 the algorithm for comparing functions is described. Section 3 is devoted to the analysis of the nature of patches, section 4 - searching for unpatched defects. Section 5 describes the implementation of the method, section 6 - related work review. Section 7 completes the article describing the results of the work.

II. The Algorithm of Executables Comparison

The algorithm of executables comparison as an input receives two executable files. An output is a set of matched functions pairs and set of corresponding instructions pairs between them. The input files are disassembled and translated

* The paper is supported by RFBR grant 18-07-01153 A

into the intermediate representation called REIL [10], which is a meta-assembly and platform independent language. It consists of 17 instructions. Each of them perform one elementary operation and does not have side effects, which eases the process of static analysis. The comparison algorithm [1] uses PDG and CG of executable files, which are generated based on REIL representation. At first, several heuristics are applied for matching functions. Then Hungarian algorithm is used to detect the most similar functions [11]. Similarity is calculated using algorithm of detecting code clones [12] [13] (based on finding maximal common subgraph of PDG). Additionally, the algorithm of executables comparison returns corresponding instructions in each matched functions.

III. Analysis of the nature of changes in new versions of executable files

After comparing two executables, the output set of matched functions and their matched instructions allows to understand which fragment of the code was added or deleted. The purpose of the algorithm is to detect important code changes that are potential fixes of defects. The list of changes considered by the tool as a defect fix is presented below.

- *New basic block is added.* A warning is issued if all instructions of a basic block from second executable file is not matched to any instructions from first executable (i.e. a new basic block is added). Adding of basic block may mean that check is added in selection statement or in a loop. The warning message includes type of the patch, addresses of the old and patched functions from the old and new versions of executables, and the address in old version of executable, after which new basic block is added.
- *New return instruction is added in a function*. A warning is issued if added code fragment adds a path to the return statement of the function. The warning message includes type of the patch, addresses of the old and patched functions from the old and new versions of executables, and the address in old version of executable, after which return statement is added.
- *Function arguments are changed.* A warning is issued if there is a path through data flow from the added instructions to the instructions that process the argument of the function. The warning message includes type of the patch, addresses of the old and patched functions from the old and new versions of executables, and the address of function call instruction in old version of executable, which arguments are changed.
- *Function call is changed*. A warning is issued if the matched functions corresponding function call instructions call different functions. As during refactoring, the names of functions may be changed, the algorithm indicates the change in the function call, if the called functions are not matched in previous step. The warning message includes type of the patch, addresses of the old and patched functions from the old and new versions of executables, and the address of function call instruction in old version of executable, which is changed.
- *Added break instruction in a loop*. A warning is issued if a new jump instruction is added in a loop, which adds control flow edge to the next instruction after loop. The warning message includes type of the patch, addresses of the old and patched functions from the old and new versions of executables, and the address of function call instruction in old version of executable, after which new break instruction is added.
- *Added continue instruction in a loop.* A warning is issued if a new jump instruction is added in a loop, which adds control flow edge to the beginning of a loop. The warning message includes type of the patch, addresses of the old and patched functions from the old and new versions of executables, and the address of function call instruction in old version of executable, after which new continue instruction is added.

IV. Searching for unpatched Defects

In this section, we consider the case when a fragment of a code with a defect was copied into several parts of a program, and a fix was done only on one clone of the buggy fragment in the new version of the program. The search of defects that remain unpatched is based on the code clone detection techniques [1] [12].

Changes in the new version of the executable file, described in the section 3, are considered as fixes. After finding the fixes, the algorithm finds clones of the buggy fragments in the new version of the executable file. Such clones may need to be fixed.

As the practice shows, fragments only with changed instructions often contain a few instructions. The search for clones of these fragments leads to numerous false positives. To solve this problem, three different algorithms have been developed for expanding buggy fragments of code in old version.

- The fragment of the buggy code is expanded until the level of the function in which it is located. There is a possibility for analyst to set the minimum percentage of similarity.
- A fragment of an buggy code is expanded by basic blocks according to the following principle: a basic block, in which a fix is found, is inserted into the set, then neighbor basic blocks are considered along the control flow. The number of basic blocks can be limited by the analyst. In the new version of the executable file, the resulting set of basic blocks is searched taking into account the operation codes and control flow between the basic blocks.
- The fragment of the buggy code is expanded by instructions according to the following principle: the instructions of the basic block in which the fix is found are inserted into the set, then the neighbor basic blocks are considered along the data flow. Next, a search of clones in the new version of the executable file is made by taking into account the data flow.

V. Structure of the tool

The fig. 1 shows the overall structure of the tool. As an input the tool receives two executable files. At first, executable files are disassembled using IDA Pro [14] (as it is a state of the art and interactive disassembler). Then, the recovered assembler is translated into REIL [10] representation using Binnavi [15]. After the translation, the PDG and CG are generated and then used for comparison two versions of program, analyzing the nature of changes and for searching the unpatched fragments.

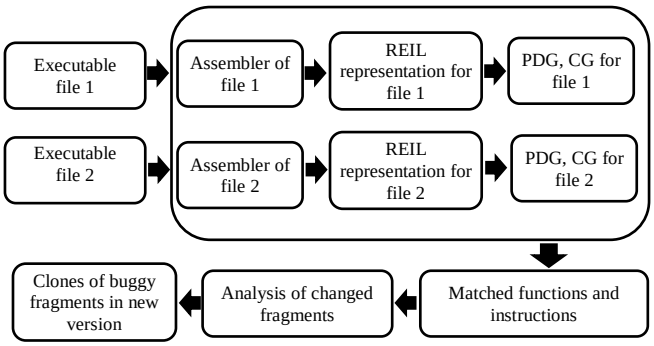

Fig. 1. Architecture of the tool.

VI. RELATED WORK

SPAIN [16] is a scalable system for binary difference analysis that automatically identifies security changes and summarizes the changed patterns as well as the corresponding defect patterns. In particular, if the initial and corrected versions of the executable program are given, SPAIN finds the functions that have been changed. The system then detects changed traces (that is, a sequence of basic blocks) for each corrected function in order to describe changes at the function level. These traces may contain safe or unsafe changes. Through semantic analysis, it determines which changes are safe and which are not. Then, by analyzing the tainted data, the samples of unsafe changes and the corresponding defects are summed up on the corrected functions.

PVDF [17] computes the semantics of corrections for defects. Then the semantics of changes is used to detect new defects in executable files. PVDF takes an executable file with a defect and a corresponding change as input data, and then retrieves the semantics. This tool is similar to SPAIN, but it assumes the presence of changes and focuses only on one specific type of defect.

Other research is based on symbolic execution. BinHunt [4] tries to find semantic differences between versions. To determine how similar the two basic blocks are, the tool checks the equality of all possible pairs of equations from both sets using SMT solvers [18]. Based on the identified semantically equal basic blocks, the reverse tracking algorithm finds the largest common subgraph between the two functions and obtains the similarity of the two functions. After locating the modified basic blocks, the information is transmitted to the analyst. iBinHunt [19] is similar to BinHunt, but extends it functionality by using interprocedural analysis and analysis of tainted data.

Neither source code nor executables of described tools are available.

VII. RESULTS

Developed tool is tested on DARPA challenge [20] and CoreBench [21] test suits. The average percent of true positives on DARPA challenge test suit is 71.3%, on Corebench test suite is 73.3%.

Table I shows the results of analyzed changes on some tests from Corebench test suite. The main reason for detecting false positives are result of incorrect comparison of executables.

TABLE I. RESULTS ON TESTS FROM THE COREBENCH SUITE

Project	Git commits		Number of found changes	True positive	Percent of true positive
	Old version	*New version*			
find	244453b8	f7197f3a	4	2	50%
find	756b47b1	24e2271e	3	2	67%
find	aca12907	b445af98	1	1	100%
grep	02f1daa1	074842d3	2	2	100%
grep	c1cb19fe	8f08d8e2	2	1	50%
grep	c2b9a4fe	6d952bee	6	6	100%
make	87ac68fe	40a49f24	5	3	60%
make	97f106fa	fc644b4c	9	7	77%
make	c3188c6f	3b1432d8	4	3	75%

Table II shows the results of tests for finding clones of the unpatched fragments in the new versions. Such fragments have been fixed in subsequent versions.

TABLE II. RESULTS WHERE UNPATCHED FRAGMENT IS DETECTED

Project	Git commits		Function name with patched defect	Function name with unpatched defect
	Old version	*New version*		
Tcpdump	b534e304	d3aae719	juniper_monitor_print	juniper_mlfr_print
Tcpdump	c2ef6938	50a44b6b	ikev1_nonce_print	1.ikev1_hash_print 2.ikev1_sig_print 3.ikev1_ke_print 4.ikev1_vid_print
Libosip	79240bdd	a54f15b8	osip_www_ authenticate_init	1.sdp_connection_init 2.osip_authorization_init 3.osip_authentication_info_init
Libosip	80a955e7	03fe3a1c	osip_negotiation_sdp_ build_offer	osip_negotiation_sdp_build_offer

ACKNOWLEDGMENT

In this article a method for analyzing the nature of program changes between its versions and searching for unpatched fragments is proposed. The method based on binary code analysis, which allows applying tool, when source code is not available partly or completely. A list of changes has been formed, which are most frequently used to correct defects. Additionally, tool detects unpatched code fragments in new version of executable. The average percentage of true positives DARPA challenge test suit is 71.3% and on Corebench test suite is 73.3%. The developed method provides an assessment of the quality of the proposed software patches.

REFERENCES

[1] H. Aslanyan, A. Avetisyan, M. Arutunian, G. Keropyan, S. Kurmangaleev and V. Vardanyan, "Scalable Framework for Accurate Binary Code Comparison," in *2017 Ivannikov ISPRAS Open Conference (ISPRAS)*, Moscow, 2017.

[2] T. Dullien and R. Rolles, "Graph-based comparison of executable objects," *Symposium sur la Securite des Technologies de l'Information et des Communications,* 2005.

[3] M. Bourquin, A. King and Ed. Robbins, "BinSlayer: Accurate Comparison of Binary Executables," in *2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, 2013.

[4] D. Gao, M. K. Reiter and D. Song, "BinHunt: Automatically Finding Semantic Differences in Binary Programs," in *Info. and Comm. Security*, 2008.

[5] "https://github.com/joxeankoret/diaphora/blob/master/doc/diaphora_help.pdf," [Online].

[6] S. C. Misra and V. C. Bhavsar, "Relationships between selected software measures and latent bug-density: Guidelines for improving quality," in *International Conference on Computational Science and its Applications, ICCSA*, Monreal, Canada, 2003.

[7] Y. Tian, J. Lawall and D. Lo, "Identifying linux bug fixing patches," *ICSE*, pp. 386-396, 2012.

[8] C. S. Corley, N. A. Kraft, L. H. Etzkorn and S. K. Lukins, "Recovering traceability links between source code and fixed bugs via patch analysis," *TEFSE*, pp. 31-37, 2011.

[9] J. Ferrante, K. Ottenstein and J. Warren, "The program dependence graph and its use in optimization," *Trans. on Prog. Lang. and Syst. (TOPLAS)*, pp. 319-349, 1987.

[10] "REIL - The Reverse Engineering Intermediate Language," Zynamics, [Online]. Available: https://www.zynamics.com/binnavi/manual/html/reil_language.htm.

[11] Harold W. Kuhn, "The Hungarian Method for the assignment problem," *Naval Research Logistics Quarterly*, vol. 2, pp. 83-97, 1955.

[12] H.K. Aslanyan, S.F. Kurmangaleev, V.G. Vardanyan, M.S. Arutunian and S.S.Sargsyan, "Platform-independent and scalable tool for binary code clone detection," *Trudy ISPRAN/Proc. ISP RAS*, vol. 1, no. 2, pp. 215-226, 2016.

[13] H. K. Aslanyan, "Effective and Accurate Binary Clone Detection," *Mathematical Problems of Computer Science*, vol. 48, pp. 64-73, 2017.

[14] "IDA Pro disassembler," Hex-Rays, [Online]. Available: https://www.hex-rays.com/products/ida.

[15] "Binnavi," Zynamics, [Online]. Available: https://www.zynamics.com/binnavi.html.

[16] Z. Xu, B. Chen, M. Chandramohan, Y. Liu and F. Song, "SPAIN: Security Patch Analysis for Binaries Towards Understanding the Pain and Pills," *IEEE/ACM 39th International Conference on Software Engineering*, pp. 462-472, 2017.

[17] S. Letian, F. Jianming, C. Jing and P. Guojun, "PVDF: An automatic Patch-based Vulnerability Description and Fuzzing method," *CSC*, pp. 1-8, 2014.

[18] R. Nieuwenhuis, A. Oliveras and C. Tinelli, "Solving SAT and SAT Modulo Theories: From an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T)," *Journal of the ACM*, p. pp. 937–977, 2006.

[19] J. Ming, M. Pan and D. Gao, " iBinHunt: Binary Hunting with Inter-procedural Control Flow," in *Kwon T., Lee MK., Kwon D. (eds) Information Security and Cryptology – ICISC 2012*, 2012.

[20] "DARPA Cyber Challenge," [Online]. Available: http://archive.darpa.mil/cybergrandchallenge/about.html.

[21] "Corebench," [Online]. Available: https://www.comp.nus.edu.sg/~release/corebench/.