Combining dynamic symbolic execution, code static analysis and fuzzing[1]

[1]*A.Yu. Gerasimov <agerasimov@ispras.ru>*
[2] *S.S. Sargsyan <sevaksargsyan@ispras.ru>*
[1] *S.F. Kurmangaleev <kursh@ispras.ru>*
[2]*J.A. Hakobyan <jivan@ispras.ru>*
[2] *S.A. Asryan <asryan@ispras.ru>*
[1] *M.K. Ermakov <mermakov@ispras.ru>*

[1] *Ivannikov Institute for System Programming, 109004, Russia, Moscow, Alexandra Solzhenitsyna, 25*
[2] *Yerevan State University, System Programming Laboratory, 0025, Armenia, Yerevan, Alex Manuugian, 1*

Abstract. This paper describes a new approach for dynamic code analysis. It combines dynamic symbolic execution and static code analysis with fuzzing to increase efficiency of each component. During fuzzing we recover indirect function calls and pass that information to the static analysis engine. This improves static path detection in the control flow graph of a program. Detected paths are used in dynamic symbolic execution to construct inputs which will cover new paths during execution. These inputs are used by the fuzzing tool to improve test-case generation and increase code coverage. The proposed approach can be used for classic fuzzing when the main goal is achieving high code coverage. As well it can be used for targeted analysis of paths and code fragments in the program. In this case the fuzzing tool accepts a set of programs addresses with potential defects and passes them to the static analysis engine. The engine constructs all paths connecting program entry point to the given addresses. Finally, dynamic symbolic execution is used to construct the set of inputs, which will cover these paths. Experimental results have shown that the proposed method can effectively detect different program defects.

Keywords: fuzzing; directed fuzzing; static analysis;, path detection; dynamic symbolic execution

DOI: 10.15514/ISPRAS-2018-30(6)-2

Для цитирования: Gerasimov A.Yu., Sargsyan S.S., Kurmangaleev S.F., Hakobyan J.A., Asryan S.A., Ermakov M.K. Combining dynamic symbolic execution and fuzzing. Trudy ISP RAN/Proc. ISP RAS, vol. 30, issue 6, 2018, pp. 25-38. DOI: 10.15514/ISPRAS-2018-30(6)-2

[1] Research is funded within the scope of RFBR grant 17-07-00702

1. Introduction

Dynamic program analysis has proven to be one of the most effective bugs finding techniques. It has a low false positive rate and most of the detected defects can be reproduced. There are several approaches for dynamic analysis. Fuzzing [1] is one of the most effective and widely used techniques, which detects defects and provides inputs to reproduce them. But it has some limitations. For example, fuzzing itself is not usable for analysis of the specific program fragments. The main reason is that inputs are randomly generated in an attempt to increase the code coverage. Dynamic symbolic execution [2] is used for systematic generation of program inputs to cover all possible execution paths. It is significantly slower than fuzzing and cannot be applied to analysis of large programs.

One of the most widely used fuzzing tools is AFL (American Fuzzy Lop) [3, 4, 5, 6]. It is a coverage guided fuzzing tool, which uses genetic algorithms for test case selection and mutation adoption. AFL can perform static instrumentation of the target program or dynamic binary code instrumentation based on QEMU [7] for coverage gathering. *LibFuzzer* [8] is an embedded fuzzing library in *LLVM* [9] compiler infrastructure, which provides the means to fuzz individual program function. *Syzkaller* [10] performs fuzzing of system functions calls for operating systems (OS) based on their descriptions. It generates and runs small programs containing system functions calls and monitors the OS state. If a crash is detected the corresponding input and generated program are stored for debugging purposes. *Peach* [11] is used for network protocol fuzzing. It introduces the concept of pit files, which describe target protocols. Grammar-based fuzzing [12] is used for fuzzing of programs (compilers, interpreters, parsers, translators etc.) accepting BNF structured inputs. It has predefined specifications for more than 120 programming languages and data formats.

Symbolic execution of a program typically refers to the process of traversing its execution tree while evaluating internal and external program data as abstract symbolic variables instead of concrete values. Program instructions applied to these variables form *path constraints* (typically represented as SMT – Satisfiability Modulo Theory – formulas). Working with these path constraints allows one to identify valuable information about multiple potential concrete execution paths at once. Dynamic symbolic execution (*DSE*) tools incorporate various techniques and improvements of basic symbolic execution to allow one to solve various practical program analysis tasks. They are widely used to perform automatic execution tree traversal by generating concrete input data. In turn, these data sets are used as test suites for defect detection and various coverage-related analyses for the target program. *Avalanche* [13, 14], *DySy* [15], *BINSEC/SE* [16] are well known *DSE* tools.

There are advantages and limitations for both fuzzing and dynamic symbolic execution. Black-box and grey-box fuzzing tools can generate a lot of inputs in a limited time, but suffer from random nature of data generation algorithm and the

only feedback which is used to support genetic algorithms is coverage data and crash/hang information for program under analysis. On the other hand, dynamic symbolic execution tools perform aggressive instrumentation of program under analysis to gather execution traces in terms of SMT formulas which drastically influences performance of program under analysis. Also, dynamic symbolic execution suffers from the *path explosion problem* [17]. Recent research focuses on combining different analysis methods to overcome limitations of methods applied separately. Amongst known solutions we want to mention jFuzz [18], Driller [19], a hybrid symbolic execution assisted fuzzing method [20] which combine fuzzing and symbolic execution to overcome known limitations of methods.

In this paper we propose an approach for combining fuzzing, dynamic symbolic execution and static code analysis for program defects detection.

2. Proposed fuzzing tool

2.1 The Architecture of the tool

The tool consists of four basic components (fig. 1). The first component is a fuzzing tool, which provides a set of mutations and basic infrastructure. The second component is a *DynamoRIO* [21] based client library for code coverage gathering. The third component is the dynamic symbolic execution tool *Anxiety* [22]. The fourth component is a program binary code static analysis engine. The proposed tool is able to perform classic fuzzing, where the main goal is to increase code coverage as much as possible. Additionally, it can perform directed analysis of the target program – instead of trying to increase code coverage the tool tries to generate input data to cover specified fragments of the target program.

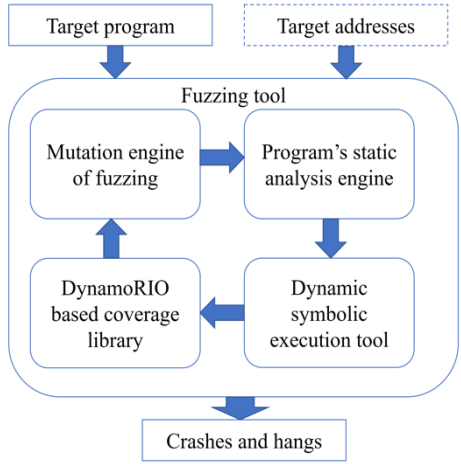

Fig. 1. The architecture of the tool

For directed fuzzing the tool accepts a set of addresses which should be executed during analysis. At first, classic fuzzing is performed until coverage stops to increase for some time (controlled by user). This typically means that there are certain fragments of code which are completely inaccessible during execution (i.e. dead code) or can only be reached with an input data set with internal dependencies that are too complex for the semi-random input mutation algorithms. In order to generate these input data sets we employ dynamic symbolic execution guided by static analysis.

2.2 Guided dynamic symbolic execution

Anxiety, the dynamic symbolic execution tool used within the system, implements «offline» concolic execution:

- it continuously performs concrete executions *along* with symbolic execution of the target program using initial input data sets and input data sets generated by the tool;
- thus, a concrete execution for an input data set produces a symbolic path constraint for this specific data set;
- this path constraint includes a number of branch points explicitly influenced by the input data set;
- for every branch point in the path constraint an attempt is made to invert the corresponding comparison and check whether the modified path constraint is *satisfiable*;
- the process of checking for satisfiability automatically produces a different input data set which is presumed to force the execution of the program onto a different path at the corresponding branch point;
- upper and lower depth limits are used to avoid processing the same branch points (producing input data sets processed previously) and creating path constraints too large to check in a limited time.

The number of branch points is a critical factor of the analysis complexity. During guided symbolic execution certain branch points are processed in a different manner based on fuzzing goals:

- «black» lists are used to skip certain branch points which were already covered during normal fuzzing (meaning that fuzzing produced at least two different input data sets which force the program execution differently for every branch point among given);
- «white» lists are used to augment path constraints with external information – which direction at the branch point must be taken for all generated paths.

Classic fuzzing, where code coverage increase is the main goal may also be improved via DSE integration. The only difference is in the list of basic blocks passed to DSE. Static analysis detects the list of basic blocks, whose both branches

were executed and pass them to DSE as a «black» list (since no new information we will be gained by inverting such blocks).

In both cases, traces of the target program execution are stored in order to perform indirect call recovery (function pointers, virtual functions). This information is used to improve static analysis which in its turn improves the results of other components.

Static analysis is periodically invoked during fuzzing to keep the data base of the target program updated using recovered indirect call addresses. This enables mutual improvement for static and dynamic analysis. Experimental results prove the effectiveness of this approach.

2.3 Static analysis engine

The static analysis engine has two basic functionalities: detecting paths in a control flow graph and program trace analysis. In the first case the tool identifies a number of paths between two program addresses. The number of limitations are applied for optimization: path's maximum length, maximum number of usages for each basic block or a function during path construction etc. These limitations are necessary to overcome the path explosion problem. Path construction consists of two basic stages (fig. 2). The first stage filters some functions based on call graph. It uses forward and backward *BFS* (Breadth-First Search) algorithm for entry and destination addresses of a target program to determine all functions which should be included in the path detection process. In the second stage we use modified DFS (Depth-First Search) for path detection. Then we construct a «white» list for DSE. It contains all basic blocks from detected paths which have branch instructions. The «white» list is used by DSE to generate data which will cover both branches of each basic block.

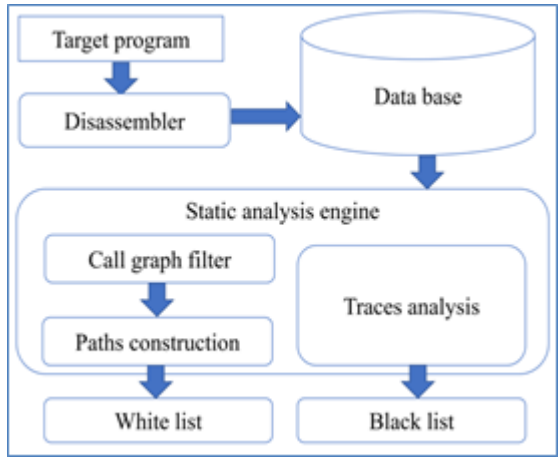

Fig. 2. Static path detection

In the second case, static analysis loads the set of traces generated by fuzzing tool and tries to find all basic blocks whose both branches were executed. Then it creates a «black» list based on these blocks to be used by DSE for optimization.

2.4 Switching metric

To switch between the fuzzing tool and DSE (static analysis included) we use a variable parameter ***N***. DSE is invoked if below formula is satisfied:

$$total\_execs - last\_effective\_exec > 10000 * N$$

where ***total_execs*** – number of executions in the moment when we try to invoke DSE, ***last_effective_exec*** – number of executions when the fuzzing last time was able to detect new execution path, ***N*** – is specified by user.
If the fuzzing tool was not able to open new execution traces for some time , then we invoke DSE.

2.5 DSE run time metric

We use special metric for calculating the maximum amount of time to allow for the DSE stage. This amount (in seconds) is calculated according to below formula:

$$runtime = 30 + total\_execs / 50000$$

where ***runtime*** – time limit for a DSE run, ***total_execs*** – number of executions at the moment when we try to invoke DSE.
The running time for DSE is at least 30 seconds (the number is determined according to experimental results). Our experiments show that less than 30 second for DSE is not enough to achieve valuable results for an average program. We increase DSE run time limit in one second after each 50.000 executions, which enables it to run longer during fuzzing.

2.6 Mutual improvement of static analysis results

While the target program is processed, static analysis engine precision is continuously improving due to indirect call address recovery. *DynamoRIO* [18] based coverage library has trace generation support, which allows us to recover actual addresses for indirect call instructions. During fuzzing process, unique traces are generated for the target program. Then they are analyzed for indirect call address recovery. The process is simple, for each executed block we store information about previously executed block. Then based on that information the actual address is recovered: if there is block in trace which belongs to some function *f* and previously executed block belongs to some function *g*, then there is an edge between *g* and *f* functions in the call graph. The newly detected edges are added to the target program data base.
Improved static analysis has positive impact on DSE results. It allows to construct more inputs which are covering different execution paths between program entry

point and destination addresses (a direct fuzzing case). These inputs improve the coverage of the fuzzing tool and improve its effectiveness. Proposed scheme of interaction between these three tools allows iteratively improve the results of each other and overall fuzzing results.

3. Results

3.1 Results of fuzzing integrated with DSE

In the table below (Tab. 1) you will find experimental results of classic fuzzing (with aim of code coverage increase) integrated with DSE. In this case we try to increase code coverage as much as possible. All detected crashes were verified manually.

Table 1. Classic fuzzing guided code coverage increase results

Operating system	Test name	Detected crashes	Running time (hours:minutes)
Debian-6.0.10	blast2	3	0:15
Debian-6.0.10	faad	1	0:20
Debian-6.0.10	efax	1	0:30
Debian-6.0.10	wavpack	5	0:30
Debian-6.0.10	tic	4	1:00
Debian-6.0.10	ul	7	1:00
Debian-6.0.10	Bsd-form	6	12:00

3.2 Results of directed fuzzing

Results of the directed fuzzing for programs from Linux distribution and DARPA [23] Cyber Grand Challenge are presented in Table 2. Static analysis has detected potential program addresses which may have defects. We run fuzzing in directed mode to generate data, which will cover specified addresses in an attempt to crash them. The last column shows the number of hits for detected address list. The fist value is the number of addresses for which the fuzzing tool was able to generate input data to cover them during execution. The second value is the number of potential buggy addresses detected by static analysis. For example, for the test ***FableReport*** static analysis has detected 15 potential defect addresses, but fuzzing tool managed to cover only 7 of them. The number of crashes is not synchronous with hit addresses due to several reasons:

- program can crash in the same address with different execution paths and fuzzing will consider it as different crashes

- if fuzzing managed to generate data which will cover specified address, it is not necessary that program should crash; the address may be false positive from static analysis or generated data do not crash it.

All results were verified manually.

Table 2. Directed fuzzing guided by static analysis results

Operating system	Test name	Crashes	Runing time (hours:minutes)	Hits
Debian-6.0.10	faad	2	21:00	1/1
Debian-6.0.10	passwd	2	0:20	1/1
Debian-6.0.10	uuenview	13	0:50	1/1
DARPA	Flash_File_System	35	2:00	1/1
DARPA	3D_Image_Toolkit	30	19:00	1/1
DARPA	Charter	9	20:00	1/1
DARPA	Diary_Parser	9	20:00	1/1
DARPA	PRU	2	1:00	1/1
DARPA	Recipe_Database	23	20:00	1/1
DARPA	SCUBA_Dive_Logging	10	20:00	1/1
DARPA	SFTSCBSISS	1	20:00	1/1
DARPA	Simple_Stack_Machine	15	20:00	1/1
DARPA	CML	10	20:00	1/1
DARPA	Eddy	9	4:00	1/1
DARPA	FablesReport	3	4:00	7/15
DARPA	Multipass3	7	4:00	1/3
DARPA	Online_job_application	4	4:00	1/1
DARPA	Overflow_Parking	2	4:00	1/1
DARPA	PTassS	5	4:00	1/2
DARPA	Sample_Shipgame	5	4:00	2/2
DARPA	SAuth	1	4:00	1/3

4. Discussion

A similar approach is used in *Badger* [24] tool. It combines fuzzing and dynamic symbolic execution in the following way: when the input is passed to symbolic execution it tries to update this input until it reaches new coverage or find a path with lower cost of analysis in terms of computational resources. This approach uses trie-based [25] symbolic execution to predict and reduce the complexity of dynamic

symbolic execution by saving a trie-like structure for path constraints gathered during path exploration until new part of path detected to execute it in symbolic manner. *Qsym* is another analysis tool [26] which combines symbolic and fuzzing. It uses optimistic solving of relaxed path constraints trying to find new paths with small cost of computations in solver and pruning conditions gathered from repetitive basic blocks from symbolic formulae to simplify constraints relying on fuzzing tool as an efficient validator of generated input.

Our approach differs from the proposed solutions. It uses static analysis to guide fuzzing and dynamic symbolic execution through continuously updated program call graph to reach destination address with the help of dynamic symbolic execution.

5. Conclusion and future work

Indirect call instructions addresses are not fully recovered based on program traces. There can be addresses, which will not be recovered because corresponding path is not executed during fuzzing. Future research directions are:

- add alias analysis on program's binary representation to improve indirect call addresses recovery;
- use available information/traces obtained from fuzzing for alias analysis improvement.

References

[1]. Fuzzing (online publication). Available at: https://en.wikipedia.org/wiki/Fuzzing, 11.12.2018

[2]. Ting Chen, Xiao-song Zhang, Shi-ze Guo, Hong-yuan Li, Yue Wu. State of the art: Dynamic symbolic execution for automated test generation. Future Generation Computer Systems, volume 29, issue 7, 2013, pp.1758-1773

[3]. American fuzzy lop (online publication). Available at: http://lcamtuf.coredump.cx/afl/, 11.12.2018

[4]. A fork of AFL for fuzzing Windows binaries (online publication). Available at: https://github.com/ivanfratric/winafl, 11.12.2018

[5]. American fuzzy lop for network fuzzing (unofficial) (online publication). Available at: https://github.com/jdbirdwell/afl, 11.12.2018

[6]. Technical «whitepaper» for afl-fuzz (online publication). Available at: http://lcamtuf.coredump.cx/afl/technical_details.txt, 11.12.2018

[7]. QEMU (online publication). Available at: https://www.qemu.org/, 11.12.2018

[8]. libFuzzer – a library for coverage-guided fuzz testing (online publication). Available at: https://llvm.org/docs/LibFuzzer.html, 11.12.2018

[9]. The LLVM Compiler Infrastructure (online publication). Available at: https://llvm.org, 11.12.2018

[10]. Syzkaller is an unsupervised, coverage-guided kernel fuzzer (online publication). Available at: https://github.com/google/syzkaller, 11.12.2018

[11]. Peach (online publication). Available at: https://www.peach.tech/products/peach-fuzzer/, 11.12.2018

[12]. Sevak Sargsyan, Shamil Kurmangaleev, Matevos Mehrabyan, Maksim Mishechkin, Tsolak Ghukasyan, Sergey Asryan. Grammar-Based Fuzzing. In Proc. of the Ivannikov Memorial Workshop. Yerevan, Armenia, 2018, pp. 32-36

[13]. Ildar Isaev, Denis Sidorov, Alexander Gerasimov, Mikhail Ermakov. Avalanche: Using dynamic analysis for automatic defect detection in programs based on network sockets. Trudy ISP RAN/Proc. ISP RAS, vol. 21, 2011, pp. 55-70

[14]. M.K. Ermakov, A.Y. Gerasimov. Avalanche: adaptation of parallel and distributed computing for dynamic analysis to improve performance of defect detection. Trudy ISP RAN/Proc. ISP RAS, vol. 25, 2013, pp. 29-38. doi:10.15514/ISPRAS-2013-25-2

[15]. Christoph Csallner, Nikolai Tillmann, Yannis Smaragdakis. DySy: Dynamic Symbolic Execution for Invariant Inference. In Proc. of the 30th international conference on Software, 2008, pp. 281-290

[16]. Robin David, Sebastien Bardin, Thanh Dinh Ta, Laurent Mounier, Josselin Feist, Marie-Laure Potet, Jean-Yves Marion. BINSEC/SE: A Dynamic Symbolic Execution Toolkit for Binary-level Analysis. In Proc. of the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), 2016, pp. 653-656

[17]. R.S. Boyer, B. Elspas, K.N. Levitt. SELECT – F Formal System for Testing and Debugging Programs by Symbolic Execution. In Proc. of the Internations Conference on Reliable software. pp. 234-245, Los Angeles, Califirnia, USA, Aprul 21-23, 1975

[18]. Jayaraman K.k, Harvison D., Ganesh V., Kiezun A. jFuzz: A Concolic Whitebox Fuzzer for Java. In Proc. of the First NASA Formal Methods Symposium, 2009, pp. 121-125

[19]. N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, G. Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In Proc. of the 2016 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA, Feb. 2016

[20]. Li Zhang, Vrizlynn L. L. Thing. A hybrid symbolic execution assisted fuzzing method. In Proc. of the TENCON 2017 - 2017 IEEE Region 10 Conference, doi: 10.1109/TENCON.2017.8227972, 5-8 Nov. 2017

[21]. DynamoRIO (online publication). Available at: http://www.dynamorio.org/, 11.12.2018.

[22]. A. Gerasimov, S. Vartanov, M. Ermakov, L. Kruglov, D. Kutz, A. Novikov, S. Asryan. Anxiety: a dynamic symbolic execution framework. In Proc. of the 2017 Ivannikov ISPRAS Open Conference, 2017, pp. 16-21. doi:10.1109/ISPRAS.2017.00010

[23]. Cyber Grand Challenge (online publication). Available at: https://www.darpa.mil/program/cyber-grand-challenge, 11.12.2018

[24]. Yannic Nolle, Rody Kersten, Corina S. Păsăreanu. Badger: complexity analysis with fuzzing and symbolic execution. In Proc. of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 322-332, Amsterdam, Netherlands — July 16 - 21, 2018

[25]. Guowei Yang, Corina S. Păsăreanu, Sarfraz Khurshid. Memoized symbolic execution. In Proc. of the 2012 International Symposium on Software Testing and Analysis, pp. 144-154 Minneapolis, MN, USA — July 15 - 20, 2012

[26]. Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, Taesoo Kim. QSYM: a practical concolic execution engine tailored for hybrid fuzzing. In Proc. of the 27th USENIX Conference on Security Symposium, pp. 745-761, Baltimore, MD, USA — August 15 - 17, 2018

Комбинирование динамического символьного исполнения, статического анализа кода и фаззинга

[1] *А.Ю. Герасимов <agerasimov@ispras.ru>*
[2] *С.С. Саргсян <sevaksargsyan@ispras.ru>*
[1] *Ш.Ф. Курмангалеев <kursh@ispras.ru>*
[2] *Д.А. Акопян <jivan@ispras.ru>*
[2] *С.А. Асрян <asryan@ispras.ru>*
[1] *М.К. Ермаков <mermakov@ispras.ru>*

[1] *Институт системного программирования им. В.П. Иванникова РАН, 109004, Россия, Москва, Александра Солженицына, 25*
[2] *Ереванский государственный университет, Лаборатория системного программирования, 0025, Армения, Ереван, ул. Алека Манукяна, 1*

Аннотация. В этой статье описывается новый подход для динамического анализа программ. Он совмещает динамическое символьное исполнение программ и статический анализ кода программ с фаззингом для повышения эффективности каждого из методов. В процессе фаззинга восстанавливаются вызовы по вычисляемым адресам и расширенный граф вызовов передается модулю статического анализа. Это позволяет улучшить вычисление путей исполнения программы в процессе статического анализа. Открытые новые пути исполнения в программе передаются модулю динамического символьного исполнения для генерации новых наборов внешних данных программы с целью исполнения и анализа программы по открытым путям исполнения. Новые наборы входных данных передаются модулю фаззинга для увеличения покрытия программы с их использованием в качестве затравки. Предложенный подход может быть использован в рамках классического алгоритма работы фаззинга с целью достижения высокого покрытия кода программы тестовыми наборами. Также предложенный метод может использоваться для направленного анализа путей и фрагментов кода программы. В этом случает фаззер формирует набор адресов и передает их модулю статического анализа. Статический анализ формирует набор путей, которые приводят к исполнению инструкций по этим адресам от точки входа в программу. Далее модуль динамическое символьного исполнения используется для построения наборов входных данных для прохождения по этим путям. Результаты экспериментов показывают высокую эффективность обнаружения программных ошибок при применении предложенного метода.

Ключевые слова: фаззинг; направленный фаззинг; статический анализ; обнаружение путей исполнения; динамическое символьное исполнение

DOI: 10.15514/ISPRAS-2018-30(6)-2

Для цитирования: Герасимов А.Ю., Саргсян С.С., Курмангалеев Ш.Ф., Акопян Д.А., Асрян С.А., Ермаков М.К. Комбинирование динамического символьного исполнения, статического анализа кода и фаззинга. Труды ИСП РАН, том 30, вып. 6, 2018 г., стр. 25-38. DOI: 10.15514/ISPRAS-2018-30(6)-2

Список литературы

[1]. Fuzzing (online publication). Режим доступа: https://en.wikipedia.org/wiki/Fuzzing, 11.12.2018
[2]. Ting Chen, Xiao-song Zhang, Shi-ze Guo, Hong-yuan Li, Yue Wu. State of the art: Dynamic symbolic execution for automated test generation. Future Generation Computer Systems, volume 29, issue 7, 2013, pp.1758-1773
[3]. American fuzzy lop (online publication). Режим доступа: http://lcamtuf.coredump.cx/afl/, 11.12.2018
[4]. A fork of AFL for fuzzing Windows binaries (online publication). Режим доступа: https://github.com/ivanfratric/winafl, 11.12.2018
[5]. American fuzzy lop for network fuzzing (unofficial) (online publication). Режим доступа: https://github.com/jdbirdwell/afl, 11.12.2018
[6]. Technical «whitepaper» for afl-fuzz (online publication). Режим доступа: http://lcamtuf.coredump.cx/afl/technical_details.txt, 11.12.2018
[7]. QEMU (online publication). Режим доступа: https://www.qemu.org/, 11.12.2018
[8]. libFuzzer – a library for coverage-guided fuzz testing (online publication). Режим доступа: https://llvm.org/docs/LibFuzzer.html, 11.12.2018
[9]. The LLVM Compiler Infrastructure (online publication). Режим доступа: https://llvm.org, 11.12.2018
[10]. Syzkaller is an unsupervised, coverage-guided kernel fuzzer (online publication). Режим доступа: https://github.com/google/syzkaller, 11.12.2018
[11]. Peach (online publication). Режим доступа: https://www.peach.tech/products/peach-fuzzer/, 11.12.2018
[12]. Sevak Sargsyan, Shamil Kurmangaleev, Matevos Mehrabyan, Maksim Mishechkin, Tsolak Ghukasyan, Sergey Asryan. Grammar-Based Fuzzing. In Proc. of the Ivannikov Memorial Workshop. Yerevan, Armenia, 2018, pp. 32-36
[13]. Исаев И.К., Сидоров Д.В., Герасимов А.Ю., Ермаков М.К. Avalanche: Применение динамического анализа для автоматического обнаружения ошибок в программах, использующих сетевые сокеты. Труды ИСП РАН, том 21, 2011, стр. 55-70
[14]. Ермаков М.К., Герасимов А.Ю. Avalanche: применение параллельного и распределенного динамического анализа программ для ускорения поиска дефектов и уязвимостей. Труды ИСП РАН, том 25, 2013, стр. 29-38. doi:10.15514/ISPRAS-2013-25-2
[15]. Christoph Csallner, Nikolai Tillmann, Yannis Smaragdakis. DySy: Dynamic Symbolic Execution for Invariant Inference. In Proc. of the 30th international conference on Software, 2008, pp. 281-290
[16]. Robin David, Sebastien Bardin, Thanh Dinh Ta, Laurent Mounier, Josselin Feist, Marie-Laure Potet, Jean-Yves Marion. BINSEC/SE: A Dynamic Symbolic Execution Toolkit for Binary-level Analysis. In Proc. of the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), 2016, pp. 653-656
[17]. R.S. Boyer, B. Elspas, K.N. Levitt. SELECT – F Formal System for Testing and Debugging Programs by Symbolic Execution. In Proc. of the Internations Conference on Reliable software. pp. 234-245, Los Angeles, Califirnia, USA, Aprul 21-23, 1975
[18]. Jayaraman K.k, Harvison D., Ganesh V., Kiezun A. jFuzz: A Concolic Whitebox Fuzzer for Java. In Proc. of the First NASA Formal Methods Symposium, 2009, pp. 121-125
[19]. N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, G. Vigna. Driller: Augmenting fuzzing through selective symbolic execution.

In Proc. of the 2016 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA, Feb. 2016

[20]. Li Zhang, Vrizlynn L. L. Thing. A hybrid symbolic execution assisted fuzzing method. In Proc. of the TENCON 2017 - 2017 IEEE Region 10 Conference, doi: 10.1109/TENCON.2017.8227972, 5-8 Nov. 2017

[21]. DynamoRIO (online publication). Режим доступа: http://www.dynamorio.org/, 11.12.2018.

[22]. A. Gerasimov, S. Vartanov, M. Ermakov, L. Kruglov, D. Kutz, A. Novikov, S. Asryan. Anxiety: a dynamic symbolic execution framework. In Proc. of the 2017 Ivannikov ISPRAS Open Conference, 2017, pp. 16-21. doi:10.1109/ISPRAS.2017.00010

[23]. Cyber Grand Challenge (online publication). Режим доступа: https://www.darpa.mil/program/cyber-grand-challenge, 11.12.2018

[24]. Yannic Nolle, Rody Kersten, Corina S. Păsăreanu. Badger: complexity analysis with fuzzing and symbolic execution. In Proc. of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 322-332, Amsterdam, Netherlands — July 16 - 21, 2018

[25]. Guowei Yang, Corina S. Păsăreanu, Sarfraz Khurshid. Memoized symbolic execution. In Proc. of the 2012 International Symposium on Software Testing and Analysis, pp. 144-154 Minneapolis, MN, USA — July 15 - 20, 2012

[26]. Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, Taesoo Kim. QSYM: a practical concolic execution engine tailored for hybrid fuzzing. In Proc. of the 27th USENIX Conference on Security Symposium, pp. 745-761, Baltimore, MD, USA — August 15 - 17, 2018