DOI: 10.15514/ISPRAS-2019-31(5)-5

Improving fuzzing performance by applying interval mutations

[1] S.S. Sargsyan, ORCID: 0000-0002-8831-4965 <sevaksargsyan@ispras.ru>
[1] J.A. Hakobyan, ORCID: 0000-0002-4094-2727<jivan@ispras.ru>
[1] H.M. Movsisyan, ORCID: 0000-0002-7582-7948 <hovhannes@ispras.ru>
[1] M.S. Mehrabyan, ORCID: 0000-0001-9846-3414 <matos@ispras.ru>
[1] V.T. Sirunyan, ORCID: 0000-0002-2213-0530 <sirunyan@ispras.ru>
[2] Sh.F. Kurmangaleev, ORCID: 0000-0002-0558-2850 <kursh@ispras.ru>

[1] Russian-Armenian University,
123 Hovsep Emin str., Yerevan, 0051, Armenia
[2] Ivannikov Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.

Abstract. This paper presents a novel approach of generation effective inputs for fuzz testing. Most applications check input format before performing basic calculations. That kind of applications usually parse service information of input file to decide whether it is supported or not. Input formats which are not supported are discarded and the application finishes its execution immediately. For example, the service information of ELF (Extensible Linking Format) file should start with the following data: "0x7f 'E' 'L' 'F'". If a file does not contain this information in header section then it will not be considered as ELF. Effective fuzzing of an application which has input validation stage is a relevant and important problem. Random changes of input files usually malform service data and the target application finishes immediately without execution of main code. This makes fuzzing process inefficient. To solve this problem, we have designed and implemented three special plugins for ISP-Fuzzer. The first plugin is intended to collect execution traces. The second plugin connects fragments of input data and executed basic blocks of the target program. Based on that information we can determine potential fragments (critical fragments) of input data which should not be mutated for new test case generation. The third plugin is designed for interval mutations. It mutates input file escaping critical fragments detected by the second plugin. Experimental results prove the effectiveness of proposed method.

Keywords: dynamic analysis; interval mutation; fuzzing.

For citation: Sargsyan S.S., Hakobyan J.A., Movsisyan H.M., Mehrabyan M.S., Sirunyan V.T., Kurmangaleev Sh.F. Improving fuzzing performance by applying interval mutations, Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 5, 2019, pp. 79-88. DOI: 10.15514/ISPRAS-2019-31(5)-5

Повышение эффективности фаззинга с помощью интервальных мутаций

[1] С.С. Саргсян, ORCID: 0000-0002-8831-4965 <sevaksargsyan@ispras.ru>
[1] Дж.А. Акоопян, ORCID: 0000-0002-4094-2727 <jivan@ispras.ru>
[1] О. М. Мовсисян, ORCID: 0000-0002-7582-7948 <hovhannes@ispras.ru>
[1] М.С. Меграбян, ORCID: 0000-0001-9846-3414 <matos@ispras.ru>
[1] В.Т. Сирунян, ORCID: 0000-0002-2213-0530 <sirunyan@ispras.ru>
[2] Ш.Ф. Курмангалеев, ORCID: 0000-0002-0558-2850 <kursh@ispras.ru>

[1] Российско-Армянский Университет,
0051, Армения, г. Ереван, ул. Овсепа Эмина 123
[2] Институт системного программирования им. В.П. Иванникова РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25

Аннотация. В статье представлен новый подход для генерации эффективных входных данных для фазз тестирования. Большинство программ перед началом выполнения основного кода проверяют формат входных данных. Часто такие приложения читают служебную информацию из входного файла и определяют поддерживается ли данный формат или нет. Входные файлы, с невалидным форматом отбрасываются. Эффективный фаззинг программ, которые проверяют служебную информацию входных данных является актуальной задачей. Мутация входных файлов часто приводит к генерации невалидной сервисной информации, и программа заканчивается до того, как исполнится ее основной код. Чтобы решить эту задачу, мы разработали и внедрили три специальных плагинов в платформу ISP-Fuzzer. Первый плагин предназначен для собирания трасс выполнения. Второй плагин связывает фрагменты входных данных с выполненными базовыми блоками целевой программы. С помощью этой информации определяются потенциальные интервалы входных данных, которые не должны мутироваться при генерации нового теста. Последний плагин разработан для интервальных мутаций. Эти мутации модифицируют входной файл, оставляя нетронутыми заданные интервалы. Эффективность предложенного метода доказана многочисленными экспериментами.

Ключевые слова: динамический анализ; интервальная мутация; фаззинг

Для цитирования: Саргсян С.С. Акопян Дж.А., Мовсисян О.М., Меграбян М.С., Сирунян В.Т., Курмангалеев Ш.Ф. Повышение эффективности фаззинга с помощью интервальных мутаций. Труды ИСП РАН, том 31, вып. 5, 2019 г., стр. 79-88 (на английском языке). DOI: 10.15514/ISPRAS-2019-31(5)-5

1. Introduction

Development of reliable software is still an essential aspect in the field of information technologies (IT). In order to improve software reliability, developers should repeatedly and constantly analyze and test their product. There are several methods and tools for that purpose [1-5]. Fuzzing is one of the most popular and efficient method of dynamic analysis. During analysis the target program is executed with mutated or generated input data. Fuzzing tool follows and verifies target programs behavior during its execution [6] (fig 1). If the target binary crashes then it reports about failure.

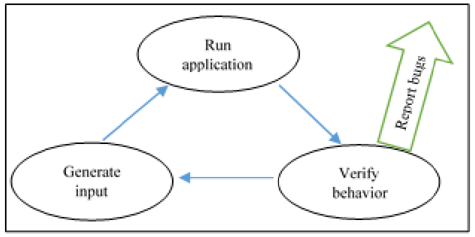


Fig 1. Process of fuzzing

There are number of state-of-the-art fuzzing tools which are able to detect faults in the target binary. One of them is AFL (American Fuzzy Lop) [7, 8, 9] – coverage based grey-box fuzzing tool. This tool has uncovered number of mistakes in list of widely known software such as: bash, OpenSSL and Mozilla Firefox [10]. There are several modifications of AFL for different tasks. WinAFL [11] is designed for fuzzing under Windows OS. kAFL [12] performs fuzzing of OS kernel. AFL uses an evolutionary algorithm to find new test data which will serve as input for next execution. For new input generation the tool uses a feedback loop to determine how much code was covered by current input. If the current input executes a new path then it is considered as interesting and saved for future mutation. Disadvantage of AFLs mutation engine is that it does not use any information about input file structure.

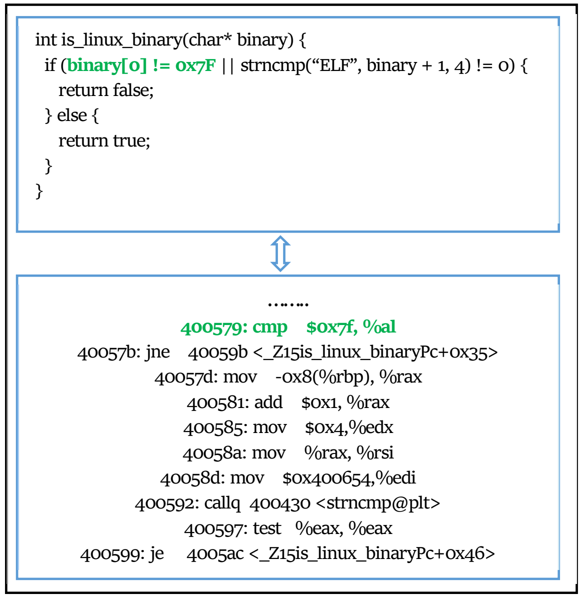
VUzzer [13] is an application aware grey box fuzzing tool which integrates static and dynamic analysis. VUzzer focuses on generation meaningful inputs which will execute new paths in target program. By static and dynamic analysis VUzzer creates a 'smart' feedback loop. Before the main fuzzing loop the tool uses static analysis to extract immediate values, magic values and other characteristic strings that affect the control flow. During the program execution, VUzzer utilize the dynamic taint analysis technique to collect information that affect the control flow branches, including specific values and the corresponding offsets. This information will be used in new test generation. VUzzer uses Pin [14], as instrumentation tool which result in a relatively slow testing speed, compared to AFL [15].

ISP-Fuzzer [16] is an extendable fuzzing framework which is implemented as coverage based, grey-box fuzzer. In order to achieve extendibility of the framework, ISP-Fuzzer provides opportunity to add custom plugins for different tasks solution. It contains number of implemented plugins, such as: BNF data generation plugin [17, 18], directed fuzzing plugin [19], DSE invocation plugin [20], etc. ISP-Fuzzer has its own mutation engine with several mutation algorithms. These algorithms, like AFL's mutation algorithms, do not know structure of input data, and may change important fragments of it such as format information in header section. For example, the first eight bytes of a PNG file contains the following values: ***'0x89 0x50 0x4E 0x47 0x0D 0x0A 0x1A 0x0A'***. If fuzzing tool changes one of these values during mutation then a mutated input will be rejected by parsing stage (i.e. fuzzing tool cannot 'dig' deeper and cover new execution paths). To address this problem, we have designed and developed a plugin for ISP-Fuzzer (ZC-DSE - Zero Cost Dynamic Symbolic Execution), which detects what intervals of input data are influencing on execution of particular basic block (BB) of the target binary. Based on that information we perform interval mutations. Interval mutations can decrease possibility of changing service bytes such as: header information, magic values, etc. These mutations are implemented as ISP-Fuzzer plugin.

The rest of this paper is structured as follow. In the section 2 we present high level overview of the proposed instrument. Section 3 describes changes in instrumentation tool for proper traces generation. The section 4 describes interaction between ZC-DSE and interval mutations plugins. In the section 5 we provide results of experimental setup for number of binary files.

2. ZC-DSE

ZC-DSE (Zero Cost Dynamic Symbolic Execution) is developed as a plugin for ISP-Fuzzer, to find what intervals of input data influence on execution of a particular BB of target binary. The tool accepts as input program's execution traces and corresponding input data. Algorithm which bounds intervals of input file to executed BB consist of two basic stages (each stage described in section A and B accordingly).



```
int is_linux_binary(char* binary) {
  if (binary[0] != 0x7F || strncmp("ELF", binary + 1, 4) != 0) {
    return false;
  } else {
    return true;
  }
}
```

```
……..
400579: cmp    $0x7f, %al
40057b: jne    40059b <_Z15is_linux_binaryPc+0x35>
40057d: mov    -0x8(%rbp), %rax
400581: add    $0x1, %rax
400585: mov    $0x4,%edx
40058a: mov    %rax, %rsi
40058d: mov    $0x400654,%edi
400592: callq  400430 <strncmp@plt>
400597: test   %eax, %eax
400599: je     4005ac <_Z15is_linux_binaryPc+0x46>
```

Fig 2. Simple example of parsing header

2.1 Selecting interesting basic blocks

At first the tool examines available traces to finds all interesting BBs in them. In order to consider BB as interesting, we use several metrics.

- *Most common*: BB considered as interesting if at least *P* percent of all traces contain it. Value of *P* by default is equal to 30% but can be adjusted by user. We presume, that frequently executed BBs corresponding to some service information parsing.
- *Branch:* With this metric the algorithm observes only branching BBs which have at least *T* BB on true branch and no more than *F* on the false. Values of *T* and *F* are easily tunable via configuration file. We presume that BBs satisfying these conditions are corresponding to code, which checks service information. False branch is executed when service information malformed, otherwise true branch executes basic functionality of target application.
- *Custom*: Any other metric can be easily implemented and integrated by user.

The *"Most common"* metric is used as default based on experimental results.

2.2 Binding interesting basic blocks to values

In the second phase ZC-DSE tries to bind intervals of input data to interesting BBs which were selected on previous stage. For that purpose, the tool intersects input files (byte level intersection) corresponding to traces containing interesting BB. We suppose resulting intervals influencing on this BB execution. If interesting BB is selected by *"Most common"* metric, and intersection of corresponding input data was empty then we stop processing of this BB. Empty result of intersection proves that there is no fixed fragment of input data influencing this BB execution.

As results ZC-DSE returns json file, which contains list of interesting BB addresses and corresponding intervals of values (from input files) which influence execution of these BBs.

For example, in fig. 2 we have fragment of code which parses header of input file and decides whether input has ELF extension. Suppose ZC-DSE have got as input two traces with respective input data (fig. 3). In these traces start address of BB2 is 0x400579 and BB2 is responsible for check whether the first position of input data is '0x7F'. In both inputs the first position is the same and equal to '0x7F'. As result ZC-DSE will return a json file which content is presented in fig. 3. ZC-DSE has determined, that in order to execute BB2 an input data should contain '0x7F' value at the first position

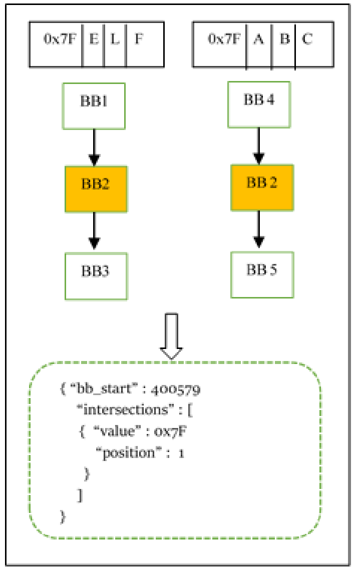


Fig. 3. Simple example of execution ZC – DSE

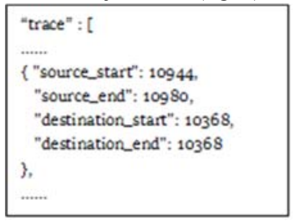
3. Traces generation

ISP-Fuzzer uses DynamoRIO [21] based client library for code coverage collection. DynamoRIO is a runtime code manipulation system which allows code transformation on any part of a program, during its execution [22]. As well it provides flexible API for code manipulation.

ISP-Fuzzer supports fork server [9] optimization, which improves fuzzing speed. DynamoRIO has special code cache for instrumented BBs. It uses already instrumented BBs for next executions. Code cache and fork server optimization are strongly connected which complicates collection of current executed trace

To collect execution traces, we should dynamically collect addresses of executed BBs. For that purpose, we should dynamically inject an assembler code in each BB while its execution. The fragment will store address of current executing BB on special buffer. Injecting assembler code in all BB is expensive and influences on fuzzing speed negatively. We use special tactic to reduce trace collection cost. Every time when the fuzzing tool executes new path, we invoke instrumentation tool with special options. This option tells DynamoRIO to inject assembler code for current trace collection and run target one more time. This time current trace is collected without affecting code cache (instrumented BBs won't be stored in it). This tactic is implemented as separate plugin.

In execution trace we keep pairs of BB addresses which were executed consequentially (Fig. 4). It enables us to construct CFG (control flow graph) of current execution. Before saving a new edge into the traces buffer we check whether it already contains that edge, to optimize buffer size. Current execution traces are stored in json files (fig. 4).



```
"trace" : [
......
{ "source_start": 10944,
  "source_end": 10980,
  "destination_start": 10368,
  "destination_end": 10368
},
......
```

Fig. 4. Example of execution trace fragment

4. Interval mutations

ISP-Fuzzer has flexible mutation engine with number of efficient mutation algorithms. These algorithms are fast but do not use information about structure of input file. For example:

1. expand data with: same/different random byte or zero byte;
2. shrink data by removing a random number of bytes;
3. modify data by setting: sequence of bytes to random values, sequence of bytes to the same random value, random range of bytes to zero values, set a random range of bytes to random non-zero values;
4. perform a word slide;
5. flip bit/byte/word;
6. generate string data by: repeating initial string a number of times, creating a list of invalid UTF-8 strings;
7. modify string data case: lower-case each character, uppercase each character and then randomly upper-case or lowercase each character;
8. modify numeric data: set to maximum/minimum value, apply arithmetic operations;
9. insert elements from dictionary;
10. concatenate random test cases from fuzzing queue.

We have developed new mutation plugin for ISP-Fuzzer (interval mutations) which allows to let intact some parts of input data. This mutation takes as input information provided by ZC-DSE plugin and does not mutate bytes of input data, which are responsible for service information checks. By default, ISP-Fuzzer invokes ZC-DSE if original mutation algorithms do not increase coverage of target binary during 3000 executions (this number is chosen after a large number of experiments). This metric is configurable. For example, a user (analytic) can set new configuration to invoke ZC-DSE:

1. if previous *X* executions were not able to detect at least *Y* new paths;
2. if previous *X* executions were not able to execute at least *Y* new BBs.

5. Experimental results

We have evaluated proposed method on several binaries from Ubuntu 18.04.2 LTS. In table 1 we present some interesting difference between default mode of ISP-Fuzzer compared with ZC-DSE plugin. Results show that plugin ZC-DSE allows to detect more paths, which in its turn allows to find more crashes and hangs. All detected crashes and hangs are manually verified.

Table 1. Results for experimental evaluation of ISP-Fuzzer with ZC-DSE

Program name	ISP - Fuzzer		ISP -Fuzzer + ZC-DSE		Difference	
	Found paths	**Found faults**	**Found paths**	**Found faults**	**Paths**	**Faults**
readelf	1175	0	2081	1	+906	**+1**
djpeg	48	0	45	1	-3	+1
objdump	391	0	407	0	+16	0
ar	11	0	9	0	-2	0
latex	80	0	196	1	+116	+1
optipng	465	34	504	41	+39	+7
gif2png	309	10	366	37	+57	+27
tiff2ps	166	0	187	0	+21	0
tiff2bw	54	0	63	0	+9	0
tiff2pdf	88	0	109	0	+21	0
tiff2rgba	37	0	44	0	+7	0
jasper	4	0	4	0	0	0
zipclock	91	0	108	1	+17	+1
dvi2tty	391	0	459	0	+68	0
strings	7	0	71	0	+64	0
pdftk	9	0	11	0	+2	0

6. Conclusion

Three plugins are developed in ISP-Fuzzer framework to allow interval fuzzing. The first plugin serves to collect execution traces, which will be used by the second plugin. The second plugin (ZC-DSE) is intended to bind interesting BBs of target program with values in input data. The third plugin uses information of the second one to mutate intervals of input data. Interval mutations can be used for binaries fuzzing accepting structured data. Experimental results show effeteness of the proposed method.

References / Список литературы

[1]. V.P. Ivannikov, A.A. Belevantsev, A.E. Borodin, V.N. Ignatiev, D.M. Zhurikhin, A.I. Avetisyan. Static analyzer Svace for finding defects in a source program code. Programming and Computer Software, vol. 40, issue 5, 2014, pp 265–275.

[2]. Hayk Aslanyan, Sergey Asryan, Jivan Hakobyan, Vahagn Vardanyan, Sevak Sargsyan, Shamil Kurmangaleev. Multiplatform Static Analysis Framework for Programs Defects Detection. In Proc. of the 11th International Conference on Computer Science and Information Technologies, 2017, pp. 315-318.

[3]. H. Aslanyan, A. Avetisyan, M. Arutunian, G. Keropyan, S. Kurmangaleev and V. Vardanyan. Scalable Framework for Accurate Binary Code Comparison, In Proc. of the 2017 Ivannikov ISPRAS Open Conference, 2017, pp. 34-38.

[4]. M. Arutunian, H. Aslanyan, V. Vardanyan, V. Sirunyan, S. Kurmangaleev, and S. Gaissaryan. Analysis of Program Patches Nature and Searching for Unpatched Code Fragments. In Proc. of the 2019 Ivannikov Memorial Workshop (IVMEM), 2019, pp. 53-56.

[5]. Aslanyan H.K. Plarform for interprocedural static analysis of binary code. Trudy ISP RAN/Proc. ISP RAS, vol. 30, issue 5, 2018. pp. 89-100. doi: 10.15514/ISPRAS-2018- 30(5)-5.

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

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

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

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

[10]. Michał Zalewski. The bug-o-rama trophy case. Available at: http://lcamtuf.coredump.cx/afl/#bugs, accessed 11.12.2018.

[11]. WinAFL - A fork of AFL for fuzzing Windows binaries (online publication). Available at https://github.com/googleprojectzero/winafl, accessed 11.12.2018.

[12]. Schumilo, S, Aschermann C, Gawlik R, Schinzel S, Holz T. kAFL: Hardware-assisted feedback fuzzing for OS kernels. In Proc. of the 26th USENIX Security Symposium, 2017, pp. 167–182.

[13]. Rawat S., Jain V., Kumar A., Cojocar L., Giuffrida C., Bos H. Vuzzer: Application-aware evolutionary fuzzing. In Proc. of the Network and Distributed System Security Symposium, 2017, 14 p.

[14]. Luk C-K., Cohn R., Muth R., Patil H., Klauser A., Lowney G., Wallace S., Reddi V.J., Hazelwood K. Pin: building customized program analysis tools with dynamic instrumentation. ACM SIGPLAN Notices, vol. 40, issue 6, pp. 190–200.

[15]. Jun Li, Bodong Zhao, Chao Zhang. Fuzzing: a survey (online publication). Available at: https://cybersecurity.springeropen.com/articles/10.1186/s42400-018-0002-y, accessed 11.12.2018

[16]. S. Sargysan, J. Hakobyan, M. Mehrabyan, M. Mishechkin, V. Akozin, Sh. Kurmangaleev. ISP-Fuzzer: Extendable fuzzing framework. In Proc. of the 2019 Ivannikov Memorial Workshop (IVMEM), 2019, pp. 68-71

[17]. S. Sargsyan, Sh. Kurmangaleev, M. Mehrabyan, M. Mishechkin, T. Ghukasyan, S. Asryan. Grammar-based Fuzzing. In Proc. of the 2018 Ivannikov Memorial Workshop (IVMEM), 2018, pp. 32-36,.

[18]. Terence Parr. The Definitive ANTLR Reference. Pragmatic Bookshelf, 2013, 328 p.

[19]. S. Sargsyan, Sh. Kurmangaleev, J. Hakobyan, H. Movsisyan, M. Mehrabyan, S. Asryan. Directed Fuzzing Based on Program Dynamic Instrumentation. In Proc. of the 2019 International Conference on Engineering Technologies and Computer Science , 2019, pp. 30-33.

[20]. 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.

[21]. DynamoRIO dynamic instrumentation tool platform, Feb. 2009. Available at http://dynamorio.org, accessed 11.12.2018.

[22]. Derek Bruening. Efficient, Transparent, and Comprehensive Runtime Code Manipulation. Ph.D. Thesis, MIT, September 2004.

Информация об авторах / Information about authors

Севак Сеникович САРГСЯН – научный сотрудник, преподаватель, заведующий кафедрой, кандидат физико-математических наук. Сфера научных интересов: анализ программ, динамический анализ кода, фаззинг.

Sevak Senikovich SARGSYAN – researcher, lecturer, head of department, Ph.D in physical and mathematical sciences. Research interests: program analysis, dynamic analysis of code, fuzzing.

Дживан Андраникович АКОПЯН – научный сотрудник, преподаватель, аспирант. Сфера научных интересов: анализ программ, динамический анализ кода, фаззинг.

Jivan Andranikovich HAKOBYAN – researcher, lecturer, PhD student. Research interests: program analysis, dynamic analysis of code, fuzzing.

Оганес Мушегович МОВСИСЯН – научный сотрудник, магистр. Сфера научных интересов: анализ программ, динамический анализ кода, фаззинг.

Hovhannes Musheghovich MOVSISYAN – researcher, master. Research interests: program analysis, dynamic analysis of code, fuzzing.\ Матевос Саргисович Меграбян – научный сотрудник, магистр. Сфера научных интересов: анализ программ, динамический анализ кода, фаззинг.

Matevos Sargisovich MEHRABYAN – researcher, master. Research interests: program analysis, dynamic analysis of code, fuzzing.

Ваагн Телемакович СИРУНЯН – научный сотрудник, бакалавр. Сфера научных интересов: анализ программ, динамический анализ кода, фаззинг.

Vahagn Telemakovich SIRUNYAN – researcher, bachelor. Research interests: program analysis, dynamic analysis of code, fuzzing.

Шамиль Фаимович КУРМАНГАЛЕЕВ – старший научный сотрудник, кандидат физико-математических наук. Сфера научных интересов: анализ программ, динамический анализ кода, фаззинг.

Shamil Faimovich KURMANGALEEV – Senior researcher, Ph.D in physical and mathematical sciences. Research interests: program analysis, dynamic analysis of code, fuzzing.