Platform for interprocedural static analysis of binary code

H.K. Aslanyan <hayk@ispras.ru>
Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia

Abstract. This paper describes the developed platform for static analysis of binary code. The platform is developed based on interprocedural, flow-sensitive and context-sensitive analysis of the program. The machine-independent language REIL is used as an intermediate representation. In this representation basic data flow analyzes are developed and implemented - reaching definitions analysis, construction of DEF-USE and USE-DEF chains, analysis for deletion of dead code, value analysis, taint analysis, memory analysis and etc. The implemented approach for functions' annotations allow propagating data between function calls, thereby making the context-sensitive analysis. The platform provides an API for using all implemented analyzes, which allows adding new analyzes as plugins.

Keywords: static analysis; binary code analysis; interprocedural analysis

DOI: 10.15514/ISPRAS-2018-30(5)-5

For citation: 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

1. Introduction

Software developers always strive to create high-quality software, meaning that it should be reliable, safe and easy to maintain. However, with increasing size and complexity of projects, the developed code contains more errors [1]. Fixing those errors can be done at any phase of the software development life cycle. Ideally, all errors are detected during the testing phase. Error detection at the later phases or after deployment may cause many difficulties. Moreover, erroneous software may result in money loss. However, even a very thoroughly tested software sometimes contains errors. Currently, various code analysis tools are widely used to detect these errors.

Static code analysis is one of the common defect detection approaches. Static analysis examines examining a code without executing a program. Through a complete analysis of syntax, semantics, control and data flow, static code analysis can find errors that are difficult or impossible to find during testing, especially on rarely executed paths. Static analysis is based on methods and approaches both from fundamental and applied research.

There are lots of approaches for static analysis of source code [2–8]. However, static analysis of executables is less studied, despite the fact that it has several advantages over the source code analysis. The first advantage of the binary code analysis compared to the source code analysis is the fact that the source code is not always available. The second advantage is that aggressive compiler optimizations may create defects in binary code that were non-existent in the source, and it is very hard to prove the optimization correctness [9-10]. The third advantage is the undefined semantics of certain language constructs that may create difficulties for a static analyzer. For example, in C/C ++ the order in which actual function parameters are evaluated is implementation defined, which can lead to false positive reports in the source code analyzer.

A production quality static analysis tool should have the following features: interprocedural analysis support, flow sensitivity, path sensitivity. In addition, a high-quality analyzer should be able to analyze large files (binary file sizes can reach hundreds of megabytes) in a few hours, provide high accuracy (a small number of false positives), and it should be easy to extend for supporting new error types.

2. Platform architecture

The proposed tool architecture was developed taking into account the following requirements:

- target architecture independent;
- context-sensitive interprocedural analysis with flow-sensitive intraprocedural analysis;
- scalability: analyzing tens of megabytes of executable files in a few hours;
- easy platform extension.

The first step is producing assembler code from an executable. Assembler language instructions are created by a disassembler using the object code as input. The tool uses the IDA Pro [11] disassembler since it supports many executable file formats for a large number of processors, automatically restores control flow graphs and call graphs. The disassembler also restores calling conventions. Then the resulting assembly code is transferred to the Binnavi [12] tool, which converts it to the REIL representation (Reverse Engineering Intermediate Language) [13]. REIL representation is an intermediate low-level language that can be used to write platform-independent analysis algorithms. It has only 17 instructions. Each instruction calculates no more than one result and has no side effects (flag settings, etc.). REIL representation is created for a virtual processor with unlimited memory and an unlimited number of registers denoted as t0, t1, t2, etc. Target machine registers can be also accessed in REIL. Fig. 1 shows the scheme for getting the assembler and REIL representation.

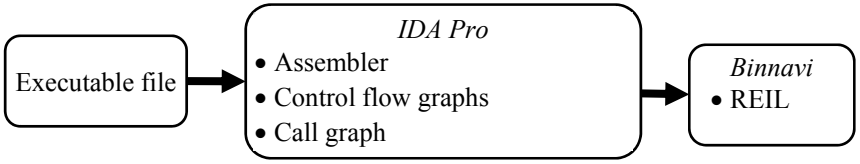

Fig. 1.Getting a REIL representation

3. Function summaries and interprocedural analysis

After generating a REIL representation, the call graph is made acyclic. First, the classical Tarjan approach [14] is used to find strongly connected components (SCCs). Second, directed cycles are identified, and an arbitrary edge is removed from each of them. This process breaks the connectivity properties of the SCCs.

Then call graph nodes are divided into groups (fig. 2) as follows: the first group has nodes that have no outgoing edges. The second group includes nodes whose descendants are in the first group. Thus, each subsequent group includes the nodes that have their successor nodes processed as belonging to the previous groups. Since the call graph has no more directed cycles, the algorithm will be completed in a finite number of steps, and each node will fall into a certain group.

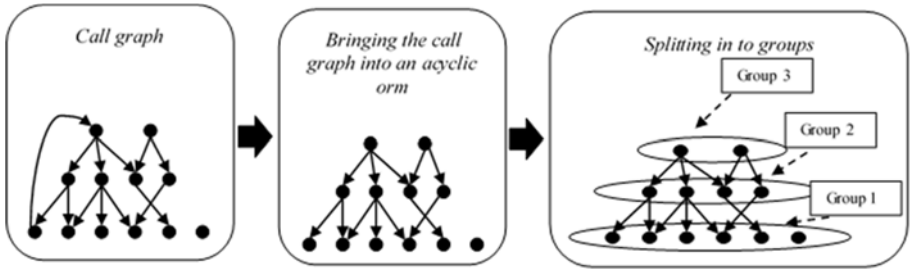

Fig. 2. Splitting nodes of the call graph into groups

Next, call graph traversal is performed according to the node groups built. Intraprocedural analyses are run starting from the first group's nodes, and each next group is only considered if the functions corresponding to all previous group's nodes have been analyzed. It should be noted that the analysis is performed only for functions with available bodies, i.e. functions from dynamic libraries are not analyzed (only summaries are available for such functions). When interprocedural analyses are completed, so-called function *summaries* are saved (summaries contain function-specific data calculated by the analyses). For example, a function returns the value that is user-controlled (like e.g. `gets` function in C/C++), or a function frees the memory pointed to by the first parameter. When analyzing a function, its callees' summaries are used. Obviously, in the absence of recursive calls, all called functions'

summaries are available. In the case of recursive calls, some edges are removed from the call graph and thus some called function summary may not be available. Such cases are handled as calls of unknown functions (without a summary). We have used the C standard library summaries from the Svace tool [2]. Also, summaries can be extended with new types of data in our platform.

Intraprocedural analyses for each function are run only once, which allows achieving scalability w.r.t. the number of functions. Splitting call graph nodes into groups gives the advantage of analyzing the functions within each group in parallel.

4. Intraprocedural analysis

Basic intraprocedural analyses that form the platform contents are performed using the REIL representation. Function summaries are used when processing function calls, and the analysis data is evaluated taking into account actual call parameters and calling conventions. This process makes the analysis context-sensitive. Currently, value analysis, reaching definitions analysis, DEF-USE and USE-DEF chains construction, dead code removal, liveness analysis, taint analysis, and dynamic memory analysis are implemented. The intraprocedural analysis architecture makes it easy to extend the set of analyses (fig. 3) and to add plugins.

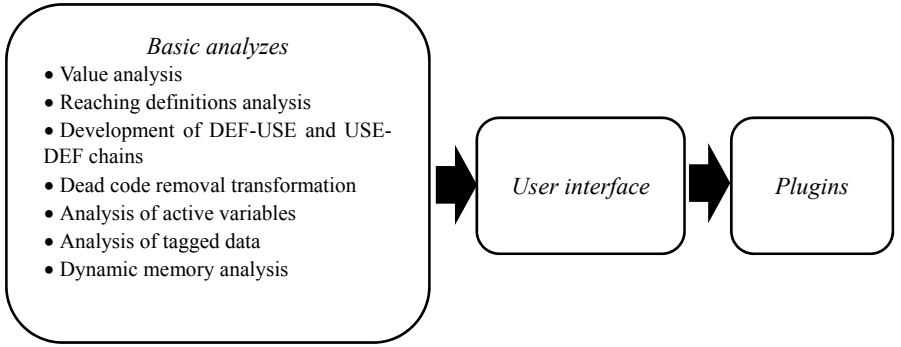

Fig. 3. Intraprocedural analysis architecture

2.1 Value analysis

Value analysis is used to track values in registers and memory cells. All registers (target architecture registers and temporaries) and all memory cells that are used in the program are called *variables*. During the analysis process all variables get values for all program points. For values stored in memory, a memory model, which tracks memory accesses for stack, heap, and static memory areas, was developed and implemented.

Value analysis is implemented based on a classic iterative data flow approach [15]. For this purpose, a semilattice is defined, that is, initial values for all variables are

specified and transfer functions are defined. All other analyses are based on the value analysis.

4.1.1 Value types

The developed value analysis has several symbolic value types: an integer type, a target architecture register, a temporary REIL register, a memory area, and a special values bottom and top. The bottom value is assigned to variables that have unknown value (the lowest element in the semilattice), and the top value is assigned to variables that may have any value (the uppermost element in the semilattice). Fig. 4 shows the value analysis semilattice.

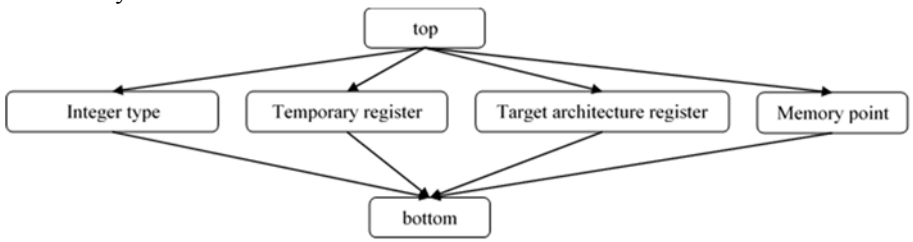

Fig. 4. Diagram of the value analysis semilattice

4.1.2 Memory model

A simple memory model is just a byte array. Memory stores and loads in this model are emulated as stores or loads to the corresponding array element. However, such a simple model has some drawbacks. It is impossible to determine concrete addresses for certain memory areas, e.g. those that are heap allocated. Moreover, function calls sequence may change during each subsequent program run, which will generally result in the ambiguity of memory values.

For proper analysis, the tool must separate different memory areas. To address the challenge, the following memory model is proposed. Memory is addressed as follows: `*(reg + constants_array) + constant`, where `reg` is a register, `constants_array` is an array of constant values, and `constant` is a constant value. `constants_array` and `constant` play the role of offset, and `constants_array` provides the ability to model multidimensional array elements and structure fiels. `reg` has a basic symbolic value. It is important to note that all formula elements are not necessarily needed to model the given cell.

- **Stack memory model.** Since it is impossible to determine the precise value of the function stack top statically, the model refers to local variables by the offset from the stack top of the current function. Therefore, the symbol `stack` for the initial address of the analyzed function's stack is used, and all local variables are modeled relative to this address. For example, in the x86 architecture, after the instruction `mov eax, esp+4` the value of `eax` will be `stack+4`, and after the instruction `mov ebx, [esp+8]` the value

of ebx will be `*(stack+{8})`. `constants_array` provides the ability to model values of structure fields. For example, if the value of the `a->b->c` expression in C code is in `ebx`, then after processing all REIL instructions the value of `constants_array` for `ebx` will be `{offset b in structure a, offset c in structure b}`.

- **Heap memory model.** To model heap memory accesses, the `heap` symbol is used and the instruction address of the memory allocation function call is put into `constants_array`. For example, after processing the `malloc` call with the address equal to `0xFFFFFFFF` (on the x86 architecture with the cdecl calling convention), `eax` will be `*(heap+{0xFFFFFFFF})`.
- **Static memory model.** Static and global variables are modeled directly with their address with or without an offset. After compilation, all static variables' addresses are known, and the variable address is put in `constants_array`, and its offset is put to `constant`.

4.1.3 Value analysis implementation

The value analysis algorithm is based on the iterative data flow approach [15]. The semilattice, transfer functions and initial values are defined. The top/bottom semilattice elements are denoted as `top`/`bottom`, respectively. `Bottom` is the initial value for all variables except stack top and function arguments.

Transfer functions are defined for REIL instructions as follows. Let us define the register value `ti` as `Val(ti)`. For example, for the instruction `add t1, t2, t3` (it adds the value `t1` to `t2` and stores in `t3`) the transfer function is defined as follows: all variables' values remain unchanged except for `t3`, and `Val(t3)` will be defined as follows:

- `top`, if `Val(t1)=top` or `Val(t2)=top;`
- `bottom`, `if Val(t1)=bottom` or `Val(t2)=bottom;`
- `Val(t1)+Val(t2)`, if `Val(t1)` and `Val(t2)` are integer constants;
- `*(reg+constants_array)+(constant+v)`, if `Val(t1)=*(reg + constants_array)+constant` and `Val (t2)` is an integer constant that is equal to `v;`
- `*(reg+constants_array)+(constant+v)`, if `Val(t2)=*(reg + constants_array)+constant` and `Val (t1)` is an integer constant that is equal to `v;`
- `*(reg)+v`, if `Val(t1)` is a register that equals to `reg`, and `Val (t2)` is an integer constant that is equal to `v;`
- `*(reg)+v`, if `Val(t2)` is a register that equals to `reg` and `Val (t1)` is an integer constant that is equal to `v;`
- `top`, if none of the above applies.

Similarly, transfer functions for other 17 REIL instructions are defined. The iterative algorithm converges as we limit the number of calculated values for each variable, so the algorithm stops when no values are changed or the above limit is reached.

4.2 Data flow analyses implementation

Based on the value analysis, other classical data flow analyses are implemented (reaching definitions analysis, dead code removal, liveness analysis, taint analysis, and dynamic memory analysis). The above analyses are also performed using the iterative data flow algorithm [15]. Semilattices and transfer functions are similarly defined, and initial values are assigned to variables. DEF-USE and USE-DEF chain construction is based on reaching definitions. The platform provides an API for working with all existing analyses, which allows implementing new analyses as plugins.

5. Related work

Balakrishnan and Reps describe in [9] an approach for analyzing value intervals. It is implemented in the CodeSurfer/x86 tool, which can be used to analyze executables for the x86 architecture. The tool uses the IDA Pro disassembler to restore the program assembly code, its control flow graphs and the call graph. The tool implements a memory model, based on which the interprocedural, context-sensitive value interval analysis is performed.

In [16] [17] [18], platforms for analyzing x86 executables are developed and implemented. These works implement an intermediate language and a disassembler, also adapting value interval analysis of [9] values for their intermediate representation. In [17], tainted data analysis is developed in addition to the above.

The paper [19] presents the BAP tool for analyzing executable files built for the x86 and ARM architectures. Both dead code removal and DEF/USE chain construction are implemented, but the analyses do not take into account memory data dependencies, which significantly lowers their quality.

The platform described in our work has two main functional advantages: it does not depend on target architecture and uses the function summary approach, which allows achieving linear scalability w.r.t. the number of analyzed functions.

6. Experimental results

All algorithms described in the paper were implemented and tested on real and artificial examples. Table 1 shows running times of all the described analyses for lepton, php and clam projects. Tests were run on a machine with a Core i5 processor, 4 cores and 16 GB RAM.

As can be seen from the table, php has a larger size compared to clam, but the analysis time of this project is shorter. Such results can be explained by the fact that functions in the clam project are much larger on average than php functions. Therefore, parallel function analysis in php is much more efficient.

Table 1. Experimental results

Executable file	Architecture	Size	The time of all analyses
lepton	x86	5 MB	19 min 21sec
php	x64	29 MB	3 h 12min
clam	x86	18 MB	4 h 20min

7. Conclusion and further work

In this work, we have presented a platform for binary code analysis that is target independent and supports a variety of classical data flow analyses. The application of the developed platform using the implemented APIs can be found in [20-24]. These projects, in particular, used reaching definitions analysis and USE-DEF/DEF-USE chains for building program dependency graphs.

In the future, we plan to add analyzers for finding critical errors in binary code. In addition, as the REIL representation does not support floating point numbers, the described analyses currently work only with integer types, and we plan to add such support, which will increase the analyzers' accuracy.

References

[1]. S. C. Misra and V. C. Bhavsar. Relationships between selected software measures and latent bug-density: Guidelines for improving quality. In Proc. of the International Conference on Computational Science and its Applications, ICCSA, Monreal, Canada, 2003.

[2]. V. P. Ivannikov, A. A. Belevantsev, A. E. Borodin, V. N. Ignatiev, D. M. Zhurikhin and A. I. Avetisyan. Static analyzer Svace for finding defects in a source program code. Programming and Computer Software, vol. 40, no. 5, 2014, pp. 265-275.

[3]. Coverity scan. Synopsys, https://scan.coverity.com/.

[4]. Klocwork static code analysis. RogueWave software, https://www.roguewave.com/products-services/klocwork/static-code-analysis.

[5]. Fortify Static Code Analyzer. Micro Focus, https://software.microfocus.com/ru-ru/products/static-code-analysis-sast/overview.

[6]. IBM AppScan. IBM, https://www.ibm.com/us-en/marketplace/ibm-appscan-source.

[7]. V. K. Koshelev, V. N. Ignatiev, A. I. Borzilov and A. A. Belevantsev. SharpChecker: Static analysis tool for C# programs. Programming and Computer Software, vol. 43, no. 4, 2017, pp. 268–276.

[8]. A. A. Belevantsev. Multilevel static analysis for improving program quality. Programming and Computer Software, 2017, pp. 321–336.

[9]. G. Balakrishnan and T. Reps. WYSINWYX: What You See Is Not What You eXecute. ACM Transactions on Programming Languages and Systems, vol. 32, no. 6, 2010, pp. 1-84.

[10]. H. J. Boehm. Threads cannot be implemented as a library. In Proc. of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation, 2005, pp. 261-268.

[11]. IDA Pro disassembler. Hex-Rays, https://www.hex-rays.com/products/ida.
[12]. Binnavi. Zynamics, https://www.zynamics.com/binnavi.html.
[13]. REIL - The Reverse Engineering Intermediate Language. Zynamics, https://www.zynamics.com/binnavi/manual/html/reil_language.htm.
[14]. R. E. Tarjan. Depth-first search and linear graph algorithms. In Proc. of the 12th Annual Symposium on Switching and Automata Theory, 1971, pp. 114 - 121
[15]. V. Aho, R. Sethi and J. D. Ullman. A formal approach to code optimization. In Proceedings of a Symposium on Compiler Optimization, 1970, pp. 86-100.
[16]. J. Kinder. Static analysis of x86 executables. Ph.D. Thesis, Technische Universitat Darmstadt, 2010.
[17]. S. Cheng, J. Yang, J. Wang, J. Wang and F. Jiang. LoongChecker: Practical Summary-Based Semi-simulation to Detect Vulnerability in Binary Code. *In* Proc. of the 10th International Conference on Trust, Security and Privacy in Computing and Communications, Changsha, 2011, pp. 150-159.
[18]. D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam and P. Saxena. BitBlaze: A New Approach to Computer Security via Binary Analysis. In Proc. of the 4th International Conference on Information Systems Security, 2008, pp. 1-25.
[19]. D. Brumley , I. Jager , T. Avgerinos and E. J. Schwartz. BAP: A Binary Analysis Platform. Lecture Notes in Computer Science, vol. 6806, 2011, pp. 463-469.
[20]. H. K. Aslanyan. Effective and Accurate Binary Clone Detection. Mathematical Problems of Computer Science, vol. 48, 2017, pp. 64-73.
[21]. G. S. Keropyan, V. G. Vardanyan, H. K. Aslanyan, S. F. Kurmangaleev and S. S. Gaissaryan. Multiplatform Use-After-Free and Double-Free Detection in Binaries. Mathematical Problems of Computer Science, vol. 48, 2017, pp. 50-56.
[22]. 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, Moscow, 2017, pp. 34-38.
[23]. H. Aslanyan, S. Asryan, J. Hakobyan, V. Vardanyan, S. Sargsyan and S. Kurmangaleev. Multiplatform Static Analysis Framework for Programs Defects Detection. In CSIT Conference 2017, Yerevan, Armenia, 2017.
[24]. H.K. Aslanyan, S.F. Kurmangaleev, V.G. Vardanyan, M.S. Arutunian, S.S.Sargsyan. Platform-independent and scalable tool for binary code clone detection. Trudy ISP RAN/Proc. ISP RAS, vol. 1, issue 2, 2016. pp. 215-226 (in Russian). DOI: 10.15514/ISPRAS-2016-28(5)-13.

Платформа межпроцедурного статического анализа бинарного кода

А.К. Асланян <hayk@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25.

Аннотация. В рамках данной статьи описывается разработанная платформа для статического анализа бинарного кода. Платформа разработана на основе межпроцедурного, потоко-чувствительного и контекстно-чувствительного анализа программы. В качестве промежуточного представления используется машинно-независимый язык REIL. На этом представлении разработаны и реализованы основные

анализы потока данных - анализ достигающих определений, построение DEF-USE и USE-DEF цепочек, трансформация для удаления мертвого кода, анализ значений, анализ помеченных данных, анализа памяти и т.д. Реализованный подход аннотации функций позволяет распространять данные между вызовами функций, тем самым сделав анализ чувствительным к контексту. Платформа предоставляет программный интерфейс для работы со всеми реализованным анализами, что позволяет добавлять новые анализы в качестве плагинов.

Ключевые слова: статический анализ, анализ бинарного кода, межпроцедурный анализ

DOI: 10.15514/ISPRAS-2018-30(5)-5

Для цитирования: Асланян А.К. Платформа межпроцедурного статического анализа бинарного кода. Труды ИСП РАН, том 30, вып. 5, 2018 г., стр. 89-100 (на английском языке). DOI: 10.15514/ISPRAS-2018-30(5)-5

Список литературы

[1]. S. C. Misra and V. C. Bhavsar. Relationships between selected software measures and latent bug-density: Guidelines for improving quality. In Proc. of the International Conference on Computational Science and its Applications, ICCSA, Monreal, Canada, 2003.
[2]. V. P. Ivannikov, A. A. Belevantsev, A. E. Borodin, V. N. Ignatiev, D. M. Zhurikhin and A. I. Avetisyan. Static analyzer Svace for finding defects in a source program code. Programming and Computer Software, vol. 40, no. 5, 2014, pp. 265-275.
[3]. Coverity scan. Synopsys, https://scan.coverity.com/.
[4]. Klocwork static code analysis. RogueWave software, https://www.roguewave.com/products-services/klocwork/static-code-analysis.
[5]. Fortify Static Code Analyzer. Micro Focus, https://software.microfocus.com/ru-ru/products/static-code-analysis-sast/overview.
[6]. IBM AppScan. IBM, https://www.ibm.com/us-en/marketplace/ibm-appscan-source.
[7]. V. K. Koshelev, V. N. Ignatiev, A. I. Borzilov and A. A. Belevantsev. SharpChecker: Static analysis tool for C# programs. Programming and Computer Software, vol. 43, no. 4, 2017, pp. 268–276.
[8]. A. A. Belevantsev. Multilevel static analysis for improving program quality. Programming and Computer Software, 2017, pp. 321–336.
[9]. G. Balakrishnan and T. Reps. WYSINWYX: What You See Is Not What You eXecute. ACM Transactions on Programming Languages and Systems, vol. 32, no. 6, 2010, pp. 1-84.
[10]. H. J. Boehm. Threads cannot be implemented as a library. In Proc. of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation, 2005, pp. 261-268.
[11]. IDA Pro disassembler. Hex-Rays, https://www.hex-rays.com/products/ida.
[12]. Binnavi. Zynamics, https://www.zynamics.com/binnavi.html.
[13]. REIL - The Reverse Engineering Intermediate Language. Zynamics, https://www.zynamics.com/binnavi/manual/html/reil_language.htm.
[14]. R. E. Tarjan. Depth-first search and linear graph algorithms. In Proc. of the 12th Annual Symposium on Switching and Automata Theory, 1971, pp. 114 - 121

[15]. V. Aho, R. Sethi and J. D. Ullman. A formal approach to code optimization. In Proceedings of a Symposium on Compiler Optimization, 1970, pp. 86-100.

[16]. J. Kinder. Static analysis of x86 executables. Ph.D. Thesis, Technische Universitat Darmstadt, 2010.

[17]. S. Cheng, J. Yang, J. Wang, J. Wang and F. Jiang. LoongChecker: Practical Summary-Based Semi-simulation to Detect Vulnerability in Binary Code. *In* Proc. of the 10th International Conference on Trust, Security and Privacy in Computing and Communications, Changsha, 2011, pp. 150-159.

[18]. D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam and P. Saxena. BitBlaze: A New Approach to Computer Security via Binary Analysis. In Proc. of the 4th International Conference on Information Systems Security, 2008, pp. 1-25.

[19]. D. Brumley , I. Jager , T. Avgerinos and E. J. Schwartz. BAP: A Binary Analysis Platform. Lecture Notes in Computer Science*,* vol. 6806, 2011, pp. 463-469.

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

[21]. G. S. Keropyan, V. G. Vardanyan, H. K. Aslanyan, S. F. Kurmangaleev and S. S. Gaissaryan. Multiplatform Use-After-Free and Double-Free Detection in Binaries. Mathematical Problems of Computer Science*,* vol. 48, 2017, pp. 50-56.

[22]. 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, Moscow, 2017, pp. 34-38.

[23]. H. Aslanyan, S. Asryan, J. Hakobyan, V. Vardanyan, S. Sargsyan and S. Kurmangaleev. Multiplatform Static Analysis Framework for Programs Defects Detection. In CSIT Conference 2017, Yerevan, Armenia, 2017.

[24]. А. Асланян, Ш. Курмангалеев, В. Варданян, М. Арутюнян и С. Саргсян, «Платформенно-независимый и масштабируемый инструмент поиска клонов бинарного кода,» Труды ИСП РАН, т. 28, № 5, pp. 215-226, 2016. DOI: 10.15514/ISPRAS-2016-28(5)-13.