Detecting Code Security Breaches by Means of Dataflow Analysis

Sergei Borzykh
Development Department
NPO Echelon, JSC
Moscow, Russia
mail@npo-echelon.com

Alexey Markov, Valentin Tsirlov
Information Security Department
Bauman Moscow State Technical University
Moscow, Russia
a.markov@bmstu.ru, v.tsirlov@bmstu.ru

Alexander Barabanov
Testing Department
NPO Echelon, JSC
Moscow, Russia
ab@cnpo.ru

***Abstract*—We discuss static and dynamic methods of the code analysis. A new approach to the static analysis method based on command flow graphs is presented. Practical cases and implementations of this dataflow approach are given.**

***Keywords*— information security; software security; static analysis; heuristic analysis; vulnerabilities; defects; production models; data-flow analysis**

I. Introduction

IT-based solutions are currently used everywhere, and significant problems are represented by both internal software errors of the information systems, and malicious source code implemented in the information system software. The consequences of both problems lead to violation of access, integrity and confidentiality of the processed information, which can result in financial and reputational losses of the business. This is a reason of growing financial losses over the last few years. High quality and failure-free operation of the source code is a burning issue of the software industry. Ever growing complexity of the software complexes, their use in the management and control systems of the government and the industrial production require continuous upgrading of the software testing and control methods [1-11].

II. Static and Dynamic Methods of the Code Analysis

Upon the whole, the testing methods used in the audit of the software systems security may be divided into two groups: static methods (structural testing) and dynamic methods (functional testing). Static methods of the code analysis, which do not require running of the analysed code for its operation, allow for full or partial automation [12, 13]. Such methods are most frequently used in case of full access to the software system and its source texts, which is called “a white-box technique”. It employs source and loading modules of the program and its component. The benefit of the static code analysis is that it does not require multiple program runs under various operational conditions (condition of the environment and input data) and possibility to achieve a greater degree of automation of the tests for the program defects based on their design features. When developing software for special-purpose informational systems, these methods are used to search for random code defects, and hidden software functionality (backdoors) [14, 15].

Dynamic software analysis is a method of analysis that stipulates program running on real or virtual processor [16]. Functional testing is most in demand during the study of the programs by black box method, when there is access to only external software interfaces without account of their structure, back-end interfaces or status. The approach is used to study accuracy and stability of the software operation within the framework of the key jobs of the test engineers, however, the method is not always effective for searching of errors related to combinations of rarely used input data, and for identifying intentional backdoors there. Static analysis of the software source texts is closely related to development of compilation systems, and many approaches of static analysis use elements of the compiler theory, namely, the code view models [17, 18].

III. Signature Analysis as the Main Method

The approach that is called signature analysis implies the search for software defects in the software code by comparing code fragments with the samples from the database of templates (signatures) of the security defects. Depending on the method for correlating fragments of the code to the template, and the intermediate representation in use, there may be algorithms of searching for a substring in the string, and query language for structured information (for instance, XQuery for XML), or specially designed methods of correlation, but in each case each of the signatures represents the decision procedure, which employs various presence bits of potentially harmful structure. [18] provides examples of the rules for generating error signatures, which correspond to the CWE standard. We can see here that the signature methods are not limited to the types of defects and are preferable, when dealing with the backdoors.

Improvement of the operational qualities of the static code analysis is mainly related to minimizing the number of “false positives” while preserving maximum fullness of the list of the types of potentially harmful structures [19]. Therefore, the instruments describing signatures of the code defects shall ensure maximum flexibility in defining a defect with account of diversity in the syntax of the programming language under study.

The field for designing means of static analysis is now actively developing: new directions of analysis do not force out the reputable approaches, on the contrary, they complement them by integrating the advantages of the predecessors. For instance, such approach as dataflow analysis may compensate for the drawbacks of the template-based code defect search, which does not allow for high quality of identification of SQL-, Path-, XSS-injections, and other types of code injections, however, it will require large RAM and computing resources of the processor [20, 21, 22].

An interesting manifestation of symbiosis of the analysis methods is when potentially harmful structures, which have been initially identified by the customary signature method is supported by the automated method using highly-specialized, costly, but efficient procedures [23-25].

IV. Dataflow analysis

The dataflow analysis can be described as a process of gathering information about the use, defining and dependency of data in the analysed program [26, 27]. The dataflow analysis uses command flow graph generated based on the code tree. This graph represents all possible paths for running this program: the nodes stand for 'linear', consecutive fragments of the code without any transitions, and the edges stand for potential transfer of control between these fragments.

Syntactic analysis allows for identifying control structures, such as procedure, function or method calls, which, in their turn, allow building call graphs, control flow graphs, and identifying assignation and the others that allow building dataflow graphs [17, 18, 28]. Control and dataflow graphs are used for analysis of the local program blocks (mainly, the content of the functions, procedures and methods - local analysis). Control flow graphs allow analysing program behaviour on a more general level (on the level of the file, module or the entire program - global analysis).

The dataflow analysis can be used for proper detection of certain types of defects (as a rule, in operation) with a minimum number of false positives: SQL-, command-, XSS-injections, other types of code injections and setting directly in the code of the authentication data. It should be noted that despite the differences in these defects, most of them implement the following defect use pattern.

1. Data is received from the user (consequently, untrusted data).
2. Data propagates through the program depending on the conditions and cycles.
3. Data is transformed, or filtered, or remains unchanged.
4. Finally, untrusted data gets access to the vulnerable function (buffer management, SQL query running etc.).

There is a mechanism for dataflow analysis called "taint propagation", which allows for identifying the defect, but also shows the data propagation path, starting from the entry point (user input), through the program and to the function vulnerability [29]. An interesting instance of such mechanism of dataflow analysis is "constant propagation" - search for authentication data (login, password, IP-address) directly in the software source code. Let us review a code fragment:

```
String login = "Some Constant";
```

Such code fragment can be sought using signature analysis (search as per templates). It only requires representation rule:

```
VARIABLE ("login" OR "password")
OPERATOR ("=") CONSTANT(*);
```

However, these code fragments can show that such code was written for debugging and remained in the final software version by accident, or was added intentionally, provided there was assurance that the code would not be inspected. If a malicious developer wants to hide the imbedded defect from the person, who inspects the code, but also from the means of static analysis, the code may be written, for instance, this way:

```
   String label = "somewhere".substring(0,4);
   String summ =
LogConstant.class().getClassName().toLowerCase()
;
   String upd_time = summ.substring(3,
summ.lenght()-3);
   Char ascii_conv = 95;
   String login = label + ascii_conv + upd_time;
```

If we break down parts of code fragment into various modules and files of source texts, it will be next to impossible to identify the defect using manual analysis, as well as many known automated methods.

The "constant propagation" mechanism of the dataflow analysis may define the values of the variables, their concatenation and transfer into other variables, and final values of the variables. As a result, the defect may be identified and, consequently, unauthorized access to the functional capabilities of the software may be prevented.

V. Operating principles and application of the dataflow analysis

Previous sections show the importance of static analysis and general issues. Following sections describe the main approach for dataflow analysis implementation and results of its implementation in static analyzer AppChecker developed by NPO Echelon. Let us introduce a set of definitions for a future shorter description of the principles and algorithms of this method operation:

- Point — a node in the control flow graph;
- Touch points (TP) (sink or critical points) — nodes in the control flow graph, which are used for calling important functionality (in the context of the identified defect);
- Entry point - nodes in the control flow graph, where new data is received from interfaces outside the analysed code;

- Untrusted data - data received from interfaces outside the analysed code and trusted zone (allied agents, users);
- Critical flow - flow from the entry point to the touch point.

Let us define the general procedure for the search for undocumented features using dataflow analysis:

1. Prepare source texts and configurations of the analysed software.
2. Use of the static analysis tools (that implement dataflow analysis) to sourced texts and configurations prepared in step 1.
3. Processing of the results of analysis:
   - Selecting suspicious dataflow paths,
   - Analysing entry points and points of untrusted data propagation,
   - Filtering false positives.
4. Drawing up the final report.

Dataflow analysis is divided into two stages. The first stage of analysis requires engineering of critical control and dataflows in the analysed software. Below is the sequence of the algorithm actions.

1. Search for the entry points of the untrusted data in the analyzed software (template-based search). This step requires a base of entry points templates formed by inspections of standard libraries and popular frameworks.
2. Search for the points that contain potentially vulnerable functions (template-based search as well).
3. For each entry point of the untrusted data, add function, method and procedure calls that are happening in this point to the control flow tree.
4. Repeat clause 3 until you reach one of the final points specified in clause 2, or until you reach a point that does not transition into other functions, procedures or methods.
5. Once control flow trees are built, identify flows that have reached potentially vulnerable functions. Consider these flows critical.

At the second stage, analyse critical control flows, their separate points (functions, procedures and methods) and identify the fact of untrusted data propagation from the entry point to the potentially vulnerable function. Below is the sequence of the algorithm actions:

1. Obtain entry point (function, procedure or method), engineer all dataflows that affect the data received from untrusted source.
2. If untrusted data after interaction with other dataflows has not changed its status, proceed to clause 3. Otherwise, complete analysis of the current critical control flow.
3. If untrusted data were transmitted at the following point of the critical control flow, proceed to clause 4. Otherwise, complete analysis of the current critical control flow.
4. If the current point is the endpoint, proceed to clause 5. Otherwise, proceed to clause 2 with a new point and new input data. Continue, until analysis of all points in the critical control flow is complete.
5. If the current point is the endpoint, and untrusted data were transmitted to the potentially vulnerable function from the first point, enter the critical control flow on the positive triggering list. Otherwise, finalize analysis of the current critical control flow.

The list obtained at the entry to the second stage of analysis is transferred to the entry of the report generator, which control interface is also present within the graphical user interface; after that the report generator based on the transferred list and database of the defect types draws up a report on the performed static analysis.

VI. Local analysis

The following description refers to dataflow implementation in static analyzer AppChecker developed by NPO Echelon. The local analysis is normally performed for a certain block of the code (which coincides with the visibility scope depending on the programming language). The local analysis assumes obtaining information about the conditions of the program in all points of the program, i.e.:

- On creating data;
- On saving data;
- On destruction of data.

The diagram of the local analysis algorithm can be seen in Figure 1.

You can optionally store, for instance, data on the value constancy (for the “constant propagation” tool), data assurance flag (for “taint propagation”), and information about the condition of the variable.

Information can be obtained from the local block in two opposite ways listed below:

1. From bottom to top: from the point susceptible to the defect make assumptions about the properties of the data transmitted into it, go up the code to the point of entry in the local area (function, procedure or method). The approach requires consideration of all options of the program run (for each branch and iteration of the cycles), which leads to “combinatorial explosion” of the information quantity, which shall be stored during analysis.
2. From top to bottom — from the point of entry of untrusted data into the program, down along the code, with available information about all of the above points of the program. This approach allows making assumptions about running separate branches of the program and engineer sequential analysis. The

drawback of this approach is difficulty in obtaining the path from the entry point to the exit point, because the only known fact is that the path exists.

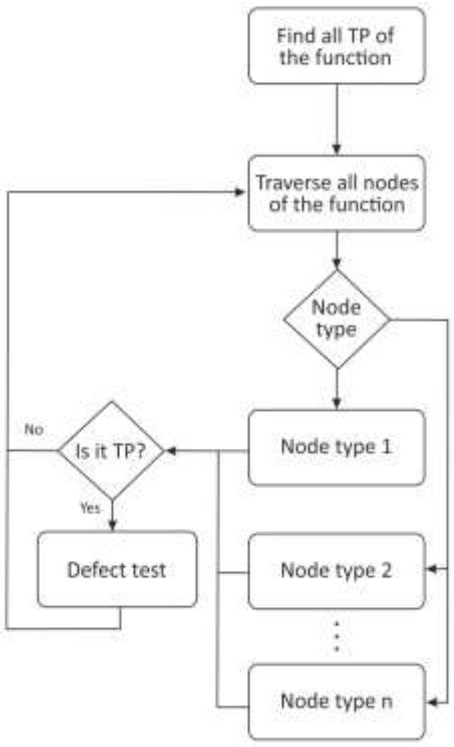

Fig. 1. Diagram of the Local Analysis Algorithm.

VII. GLOBAL ANALYSIS

Information that is available within one function (procedure, method), as a rule, is insufficient for high quality search for the defects, because many defects propagate throughout the project, or, at least one file. Global analysis is used to link data received from different functions. The global analysis engages call graphs. To ensure operation of this analysis it is sufficient to obtain certain confirmation or assumption about the properties of input and output data of separate functions in the call graph. It is important to obtain such data in the context of the functions, which are outside of the path from the point of the data entry to the point susceptible to the defects, and which analysis is necessary because the call of such functions may change the arguments or return values, which properties may depend on the properties of the input data (for instance, the substring get function, which accepts data input by the user returns taint data, although formally it is not included in the call graph).

The diagram of the global analysis algorithm can be seen in Figure 2.

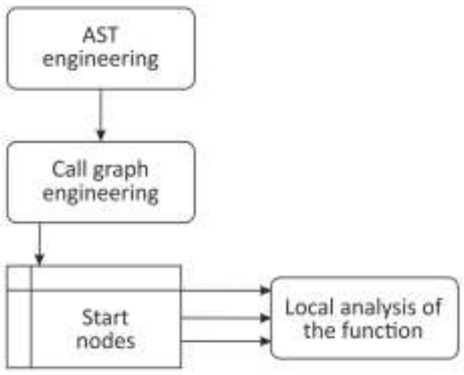

Fig. 2. Diagram of the Global Analysis Algorithm.

VIII. MATHEMATICAL DESCRIPTION OF THE DATAFLOW ANALYSIS

The ideal solution of the dataflow analysis task from the theoretical point of view consists in the search for all possible paths. Let us introduce certain symbols: B — data block for analysis, which consists from elementary subblocks $B_1, \ldots, B_n$. It is a known fact, that the dataflow values before the statement and after it are limited by the semantics of the instruction. The correlation between the dataflow values before and after the assignment statement is characterized by the transfer function. f_i shall stand for a transfer function of block B_i, which characterizes transformation of data in this block. The values of the dataflow before and after subblock B_i shall be represented as IN[B_i] (OUT[B_i] accordingly).

Suppose P is a possible execution path in the flow graph:

$$P = \text{Input} \rightarrow B_1 \rightarrow \ldots \rightarrow B_k.$$

In this case, the transfer function f_P for path P will be represented by a composition of the transfer functions $f_{k-1} \bullet \ldots \bullet f_1$. However, it should be noted that f_k is not a part of the composition, which shows that the path reached the start of subblock B_k, but not its end. Let us consider that any flow graph consists of two empty subblocks - input block, which is a start point of the graph, and output block, which is passed by all exits from the graph. The transfer functions of input and output blocks are represented by constant values.

Thus, taking into account the foregoing, the ideal solution is the array:

$$IDEAL(B) = \bigcup_P f_P(v_{input}),$$

where v_{input} is the result of the constant transfer function, which is represented by the starting input node.

It may seem that the task of the search for the ideal solution is reduced to analysis of the transfer functions f_P for all paths P in the flow graph. However, it was noted by Ullmann [17, page 724], the task of the search for the ideal solution is generally unsolvable. If block B has branches, cycles and recursions, array IDEAL[B] maybe unlimited. The assistance comes from the solution of path-based gathering [17, page 757], which is similar to the path search algorithm in the graph, so called ‘breadth first search’. This algorithm allows achieving such final number of P, that an array of all f_P covers all unique transformations of f_B.

Let us write down an iterative solution to the generalized task for the dataflow. There are two versions of such algorithm - direct and reverse. The first version proceeds from input blocks to the output, the second - goes in the reverse. The basis is Ullmann's algorithm [17, page 754].

Direct version of the algorithm:

```
OUT[INPUT] = $v_{input}$;
For (each base block B, which differs from input)
    OUT[B] = InitDataConst;
while (changes are entered in OUT)
    for (each basic block B, which differs from input)
{
 IN[B] = $\cup_{P\text{-predecessor}}$ OUT[P] ;
    OUT[B] = $f_B$ (IN[B]);
    }
Reverse version of the algorithm:

IN[INPUT] = voutput;
for(each basic block B, which differs from output)
    IN[B] = InitDataConst;
while (changes are entered in IN)
    for(each basic block B, which differs from output)
{
    OUT[B] = $\cup_{P\text{-predecessor } B}$IN[P] ;

    IN[B] = $f_B$ (OUT[B]);
    }
```

Subject to [17], if algorithm converges, its result is the solution to the dataflow problem. The obtained solution turns out to be a so called maximum fixed point, which has the property that in any other solution *IN[B]* and *OUT[B]* are already present in this solution. If in this case the analysed block is final, the convergence of the algorithm is guaranteed. These statements are proved by Jeffrey Ullmann in [17, pages 754-755].

It should be noted that in practice it is inadvisable to analyse all data used by the program. For example, if we consider unfiltered user input as input data vinput, we are going to be interested in B, where *OUT[B]* are entered in the database or output in HTML-context. Block B may also be represented by the function of the input information filtering, thus finalizing the path and marking it as safe.

IX. Examples and results

Dataflow analysis is widely spread in compilers and some sort of program analysis tools in order to find mistakes, typos and other accidentally inserted source code errors or weaknesses. This paper is dedicated to the implementation and usage of well-known analysis approach for detecting potentially harmful code areas deliberately inserted into source code. The paper subject novelty is in joint usage dataflow and signature template-based analysis for detection both embedded malicious code (backdoors, trapdoors, hard-code credentials) and weaknesses caused by accidental developer's mistakes.

Below are the examples of potentially harmful structures detected by the method described here using AppChecker software. These examples are real but quite simple because we think it is unacceptable to provide big and complex examples in this article.

1. Potential SQL-injection is identified in Dolibarr project, in *htdocs/admin/menus/edit.php* file:

B_{284} = «`$sql = "SELECT m.rowid, m.mainmenu, m.level, m.langs FROM ".MAIN_DB_PREFIX."menu as m WHERE m.rowid = ".$_GET['menuId'];`»

B_{285} = «`$res = $db->query($sql);`»

Data received from the user is entered in $sql variable, and the value of the variable without filtration is entered in SQL-request, which may lead to running of random SQL code. The critical point is string B_{285}; constant string is concatenated with taint data, and as a result the part of the string to the right of concatenation becomes taint.

2. The use of passwords set directly in the software code is identified in AWCM project, in *connect.php* file:

B_3 = «`$db_hostname = "localhost";`»

B_4 = «`$db_username = "root";`»

B_5 = «`$db_userpass = "123456";`»

B_6 = «`$db_database = "awcm";`»

B_{24} = «`@mysql_connect($db_hostname, $db_username, $db_userpass);`»

Parameters, including the password, set directly in the code, are used to connect to the database. The critical point is string B_{24}; in string B_5 the right part of the expression is a constant; in practice, the string is allocation of constant value to the variable used further to set the password. Nowadays AppChecker, which implements algorithms of signature analysis using flow analysis, contains the total of 253 rules for the search of defects in the software code in four programming languages: C/C++, Java, PHP, C#; the rules allow identifying 113 types of defects [28, 30]. AppChecker was tested in 90 projects with open source codes.

X. Conclusion

The following conclusions can came from the results of the study:

1. Based on well-reputed signature analysis approach, the suggested method of the dataflow analysis can minimize the number of false positives and simplify the development of signatures for an analyser production model.

2. The suggested method and tools will be useful for the accredited testing laboratories as well as developers of safe software tools. Secure software development practices (we would, first of all, like to mention a recently approved national standard in this field [31-33]), are being implemented at a growing rate nowadays, therefore integration of the structured testing procedure in the process of the automated system development based on static signature analysis is a high-priority task.

References

[1] D.Yu. Volkanov, V.A. Zakharov, D.A. Zorin, V.V. Podymov, I.V. Konnov, “A Combined Toolset for the Verification of Real-Time Distributed Systems,” Program. Comput. Softw., vol. 41, no. 6, pp. 325-335, November 2015. DOI:10.1134/S0361768815060080.

[2] I. S. Zakharov, M. U. Mandrykin, V. S. Mutilin, E. M. Novikov, A. K. Petrenko, and A. V. Khoroshilov, “Configurable toolset for static verification of operating systems kernel modules,” Program. Comput. Softw., vol. 41, no. 1, pp. 49-64, January 2015. DOI: 10.1134/S0361768815010065.

[3] I. S. Anureev, I.V Maryasov, and V.A. Nepomniaschy, “C-programs verification based on mixed axiomatic semantics,” Autom. Control Comput. Sci., vol. 45, no. 7, pp. 485-500, 2011. January 2012.

[4] P. N. Devyanin, A. V Khoroshilov, V. V Kuliamin, A. K. Petrenko, and I. V Shchepetkov, “Formal Verification of OS Security Model with Alloy and Event-B BT - Abstract State Machines, Alloy, B, TLA, VDM, and Z: 4th International Conference, ABZ 2014, Toulouse, France, June 2-6, 2014. Proceedings,” Y. Ait Ameur and K.-D. Schewe, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 309-313.

[5] D. Beyer and A. K. Petrenko, “Linux Driver Verification BT - Leveraging Applications of Formal Methods, Verification and Validation. Applications and Case Studies: 5th International Symposium, ISoLA 2012, Heraklion, Crete, Greece, October 15-18, 2012, Proceedings, Part II,” T. Margaria and B. Steffen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 1-6.

[6] E. di Bella, I. Fronza, N. Phaphoom, A. Sillitti, G. Succi, and J. Vlasenko, “Pair Programming and Software Defects--A Large, Industrial Case Study,” IEEE Transactions on Software Engineering, vol. 39, no. 7. pp. 930-953, Jul. 2013.

[7] S. M. Avdoshin and E. Y. Pesotskaya, “Software risk management,” 2011 7th Central and Eastern European Software Engineering Conference (CEE-SECR). pp. 1-6, 2011.

[8] A. S. Kamkin and M. M. Chupilko, “Survey of modern technologies of simulation-based verification of hardware,” Program. Comput. Softw., vol. 37, no. 3, pp. 147-152. May 2011.

[9] G. Reber, K. Malmquist, A. Shcherbakov. “Mapping the application security terrain,” Voprosy kiberbezopasnosti *[Cybersecurity Issues]*, No 1, pp. 36-39. January 2014. DOI: 10.21681/2311-3456-2014-2-36-39.

[10] A. Kozachok, M. Bochkov., T. M. Lai and E. Kochetkov. “First Order Logic for Program Code Functional Requirements Description,” Voprosy kiberbezopasnosti *[Cybersecurity issues].* No 3(21), pp. 2-7. August 2017. DOI: 10.21681/2311-3456-2017-3-2-7.

[11] E. G. Vorobiev, S. A. Petrenko, I. V. Kovaleva, I. K. Abrosimov. “Organization of the entrusted calculations in crucial objects of informatization under uncertainty,” The 20th IEEE International Conference on Soft Computing and Measurements (SCM 2017), pp. 299 - 300. May 2017. DOI: 10.1109/SCM.2017.7970566.

[12] A. Cox, B.-Y.E. Chang X. Rival. "Automatic Analysis of Open Objects in Dynamic Language Programs," International Static Analysis Symposium, Static Analysis, pp. 134-150, September 2014. DOI: 10.1007/978-3-319-10936-7_9.

[13] G. Balatsouras, Y. Smaragdakis. "Structure-Sensitive Points-To Analysis for C and C++", International Static Analysis Symposium, Static Analysis, pp. 84-104, September 2016. DOI: 10.1007/978-3-662-53413-7_5.

[14] F. Zhu, J. Wei. “Static analysis based invariant detection for commodity operating systems,” Computers and Security, vol. 43, pp. 49-63, June 2014. DOI: 10.1016/j.cose.2014.02.00.

[15] M. Bradley, F. Cassez, A. Fehnker, T. Given-Wilson, R. Huuck. “High performance Static Analysis for Industry,” Electronic Notes it Theoretical Computer Science, vol. 289, pp. 3-14, December 2012. DOI: 10.1016/j.entcs.2012.11.002.

[16] P. Gonzalez-de-Aledo, P. Sanchez, R. Huuck. "An Approach to Static-Dynamic Software Analysis," Proceedings of International Workshop on Formal Techniques for Safety-Critical Systems, pp. 225-240, November, 2015. DOI: 10.1007/978-3-319-29510-7_13.

[17] A.V. Aho, M.S. Lam, R. Sethi J.D. Ullman. Compilers: Principles, Techniques, and Tools (2nd Edition). Addison Wesley; 2nd edition (September 10, 2006).

[18] A.S. Markov, A.A. Fadin, V.L. Tsirlov. “Multilevel Metamodel for Heuristic Search of Vulnerabilities in the Software Source Code,” International Journal of Control Theory and Applications. V. 9. N 30, pp. 313-320, December 2016.

[19] M. Junker, R. Huuck, A. Fehnker, A. Knapp. “SMT-based false positive elimination in static program analysis,” Proceedings of 14th International Conference on Formal Engineering Methods, Japan, Volume 7635 of LNCS. Springer, pp. 316-331, November 2012. DOI: 10.1007/978-3-642-34281-3_23.

[20] W. Choi, S. Chandra, G. Necula, K. Sen. "SJS: A Type System for JavaScript with Fixed Object Layout," International Static Analysis Symposium, Static Analysis, pp. 181-198, September 2015. DOI: 10.1007/978-3-662-48288-9_11.

[21] Z. Luo, T. Rezk, M. Serrano. “Automated code injection prevention for web applications,” Proceedings of the 2011 international conference on Theory of Security and Applications, pp. 186-204, March 2011. DOI: 10.1007/978-3-642-27375-9_11.

[22] D. Ray, J. Ligatti. “Defining code-injection attacks,” Proceeding of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 179-190, January 2012. DOI: 10.1145/2103656.2103678.

[23] S. Seo, A. Gupta, A. Sallam, E. Bertino, K. Yim. “Detecting mobile malware threats to homeland security through static analysis,” Journal of Network and Computer Applications, vol: 38 (1) pp. 43-53, February 2014. DOI: 10.1016/j.jnca.2013.05.008.

[24] W. Lee, H. Oh, K. Yi. "A Progress Bar for Static Analyzers," International Static Analysis Symposium, Static Analysis, pp. 184-200, September 2014, DOI: 10.1007/978-3-319-10936-7_12.

[25] E. Goubault, S. Putot, F. Vedrine. "Modular static analysis with zonotopes," International Static Analysis Symposium, Static Analysis, pp. 24-40, September 2012. DOI: 10.1007/978-3-642-33125-1_5.

[26] P. Calvert, A. Mycroft. "Control Flow Analysis for the Join Calculus," International Static Analysis Symposium, Static Analysis, pp. 181-197, September 2012. DOI 10.1007/978-3-642-33125-1_14.

[27] M. Madsen, A. Moller. "Sparse Dataflow Analysis with Pointers and Reachability," International Static Analysis Symposium, Static Analysis, pp. 201-218, September 2014. DOI: 10.1007/978-3-319-10936-7_13

[28] A. Markov, A. Fadin, A. Shvets, V. Tsirlov. “The experience of comparison of static security code analyzers,” International Journal of Advanced Studies, vol. 5. № 3. pp. 55-63, September 2015.

[29] D. Zhu, J. Jung, D. Song, T. Kohno, D. Wetherall “TaintEraser: protecting sensitive data leaks using application-level taint tracking,” Newsletter ACM SIGOPS Operating Systems Review archive, January 2011, Volume 45, Issue 1, pp. 142-154. DOI: 0.1145/1945023.1945039.

[30] A. Barabanov, A. Markov, A. Fadin, V. Tsirlov. “Statistics of Software Vulnerabilities Detection During Certified Testing,” Voprosy kiberbezopasnosti *[Cybersecurity Issues].* No 2(20), pp. 2-8. May 2017. DOI: 10.21681/2311-3456-2017-2-2-8.

[31] A. Barabanov, A. Markov, A. Fadin, V. Tsirlov, I. Shakhalov. “Synthesis of Secure Software Development Controls,” The 8th International Conference on Security of Information and Networks (Sochi, Russian Federation, September 08-10, 2015). SIN ‘15. ACM New York, NY, USA, pp. 93-97. September 08-10, 2015. DOI: 10.1145/2799979.2799998.

[32] A.V. Barabanov, A.S. Markov, V.L. Tsirlov. “Methodological Framework for Analysis and Synthesis of a Set of Secure Software Development Controls,” Journal of Theoretical and Applied Information Technology. V. 88. No 1, pp. 77-88, June 2016.

[33] A. Markov, A. Barabanov, V. Tsirlov V. Models for Testing Modifiable Systems. In Book: Probabilistic Modeling in System Engineering, by ed. Andrey Kostogryzov. InTech, 2018.