Journal of Engineering and Applied Sciences 12 (Special Issue 6): 7723-7730, 2017
ISSN: 1816-949X
© Medwell Journals, 2017

Usefulness of On-The-Fly and Visualization Features in Static Vulnerability Analysis

[1]Joonseon Ahn, [2]Seungcheol Shin, [1]Hyung Joon Lim and [1]Young Sub Lee
[1]School of Electronics and Information Engineering, Korea Aerospace University 76, Hanggongdaehang-ro, Deogyang-gu, Goyang-si, 412-791 Gyeonggi-do, Korea
[2]CODEMIND Corporation, 123, Digital-ro 26-gil, Guro-gu, Seoul, Korea

Abstract: We present a static vulnerability analyzer with on-the-fly and visualization features and an empirical experiment to demonstrate its usefulness. The analyzer can find security vulnerabilities or weaknesses in program source code under development in an on-the-fly style. Also, the analyzer shows program properties related to the vulnerabilities in visualized forms which can be very helpful for testers to identify false-positives and remediate the vulnerabilities. We conducted an empirical experiment where eleven testers inspect 150 sample programs. The experiment result shows the usefulness of the on-the-fly analysis and visualization compared with manual inspection and server-based vulnerability analyzer.

Key words: On-the-fly vulnerability analyzer, secure coding, static analysis, security weaknesses, compared, properties

INTRODUCTION

The safety and security of software is becoming more critical because reliance on software systems is increasing everywhere. However, you cannot perfectly prevent software defects, weaknesses or bugs, even if you design, develop and test rigorously your software according to the software development process. You need to spend a lot of time and effort to find and remove them. At this point the static analysis tools can help you. Static analysis tools are becoming employed widely especially for finding security weaknesses or vulnerabilities and helping secure coding.

Secure coding is the practice of software development to prevent the accidental or intentional introduction of security vulnerabilities. For secure coding, you need to code keeping many security weaknesses in your mind which is not that easy since there are so many kinds of weaknesses up to hundreds which must be considered. Static analysis tools can assist finding typical security weaknesses such as SQL injection, command injection, path traversal, etc. Therefore, the necessity of static analysis tools is comprehended well and the use of them is prevailing (Chess and McGraw, 2004; Fu *et al*., 2007; Srinivasan and Thambidurai, 2007).

In reality the developers who have experienced static analysis tools are still unsatisfied in some aspects. They are struggling with understanding analysis results and finding out false positives among them (Johnson *et al*., 2013; Bradley *et al*., 2012). Static analysis technology cannot completely avoid false positives which is its inherent shortcoming. You should review every defect warning one by one from the analysis results and determine if it is a false positive or not. Even when a warning is determined as a true positive you should figure out the cause of the weakness and find the origin location or the code fragment to fix. This is the hardest work for which developers spend much time.

The other dissatisfaction is about the batch style of static analysis which means that once the analysis starts you cannot review any one warning of the analysis until the completion of the analysis. In case of over millions of LOC, the time of analysis can be up to several hours. If you can review a part of the analysis result as soon as the analysis starts, since, the analysis tool emits defect warnings whenever it recognizes defects, the analysis (the tool's action) and the review (the user's action) can be carried out simultaneously. This is the on-the-fly style of static analysis.

Among industrial static analysis tools, CODEMIND® developer can provide visualized information as graphical diagrams for understanding defect warnings, determining false positives and figuring out the location of code to fix. Also, CODEMIND® developer can play on the fly that is, you can review the result as soon as the analysis starts. These additional features are expected to improve the usability of static vulnerability analysis. However, it is subjective and based on vague intuition that the

Corresponding Author: Joonseon Ahn, School of Electronics and Information Engineering, Korea Aerospace University 76, Hanggongdaehang-ro, Deogyang-gu, Goyang-si, 412-791 Gyeonggi-do, Korea

additional features are effective in practice. In this study, we tried to present some evidence and proof for our intuition.

There were many reports about comparing static analysis tools (Emanuelsson and Nilsson, 2008; Li and Cui, 2010; Mantere *et al*., 2009; Diaz and Bermejo, 2013; Charest *et al*., 2016; Ramos, 2016). They are all about the performance of tools themselves including accuracy and speed. They do not say about the usability of static analysis tools. We focus on the performance of developer's work and show the effectiveness of the CODEMIND®'s additional features. Assume that tool A and tool B spend time T_A and T_B, respectively for some analysis and a developer spends time R_A and R_B to review the analysis results and fix the defects with tools A and B, respectively. All the existing comparison reports compare only T_A and T_B and say only about the differences of tool performance. In order to compare the tool usability, we need to compare T_A+R_A and T_B+R_B. This can be a way to find out how much the above additional features contribute to the performance of the whole activity of using a static analysis tool including finding, understanding and fixing defects.

Secure coding and program vulnerability analyzers: Secure coding is the developer's practice for software security that prevents security weaknesses and vulnerabilities. Many companies and organizations are taking proactive steps to reduce or eliminate vulnerabilities before software deployment by identifying the insecure coding practices or weaknesses.

There are well-known secure coding standards and security weakness databases that expert communities or organizations such as CWE, OWASP, CERT have been publishing (CWE, 2011, 2013; Schiela, 2017). These standards and databases provide good references about what kind of weaknesses you should find from source code. Every static vulnerability analyzer can identify its own subset from them.

It is reported that for an active project of open source code software, the number of defects discovered by static analysis tools is about 1.5 per 1,000 LOC. A new project can have 5 or 10 times more and you might review thousands of defect warnings in case of over 1 M LOC project.

Static analysis technology: Static analysis is a kind of software behavior prediction technology that calculates all possible behaviors along all execution paths in some abstract way without execution. With this technology, software vulnerability analyzer can check if there exist some security weaknesses or defects in the source code. Software defects can be syntactical or semantical. Syntactical defects such as API abuse and misspelled method name can be detected by lexical and syntactic pattern matching which is relatively simple. Semantical defects such as SQL injection and memory leak can be detected by the complex semantic analysis methods like data/control/information flow analysis, memory and value analysis and their interprocedural analysis. Semantic analysis methods must be based on abstraction, abbreviation and summarization because they cannot completely capture all the concrete behaviors of software for intractability which may lead to false positives.

Usually, static analysis tools are assessed by their accuracy and speed. The accuracy has two aspects, recall ratio and precision. The recall ratio means how many defects are discovered by the tool among all the defects existing in the source code. Lower recall ratio means more false negatives that are true defects missed by the tool. The precision presents how many defect warnings are true positives among all the defect warnings discovered by the tool. Lower precision means more false positives that are not true defects but warned by the tool. The speed means how many LOCs of the source code the tool processes for a given period of time.

CODEMIND® static analysis tools: CODEMIND® static analysis tools have two models. First one is CODEMIND® CSI/CQI which is a server-based tool and used by development teams working with a source code version control system. Developers can review their source code with a defect report or through web-based interface. With web-based interface, a security manager can monitor how many warnings come out and how many warnings are not reviewed at a given time. CODEMIND® CSI/CQI gives developers the information about the kind and the source code location of a defect warning. Also, it gives the origin point of the warning and the trace information to get from the o rigin to the warning location in the source code. Figure 1 shows CODEMIND® CSI/CQI web-based user interface. Figure 2 shows the way for CODEMIND® CSI/CQI to report an XQuery Injection warning.

Second one is CODEMIND® developer which is a personal tool operating on desktop. Figure 3 shows CODEMIND® developer. It has two additional features compared to CODEMIND® CSI/CQI. First, it provides the visualized graphs for browsing and understanding the source code. The visualized graphs are package diagram, class diagram, call relation diagram, function inside graph, etc. Also, it gives the trace information of defect warnings in the form of visualized graphs. With these graphs, you can quickly find the origin locations of the defect warnings and determine if the warnings are false or true.

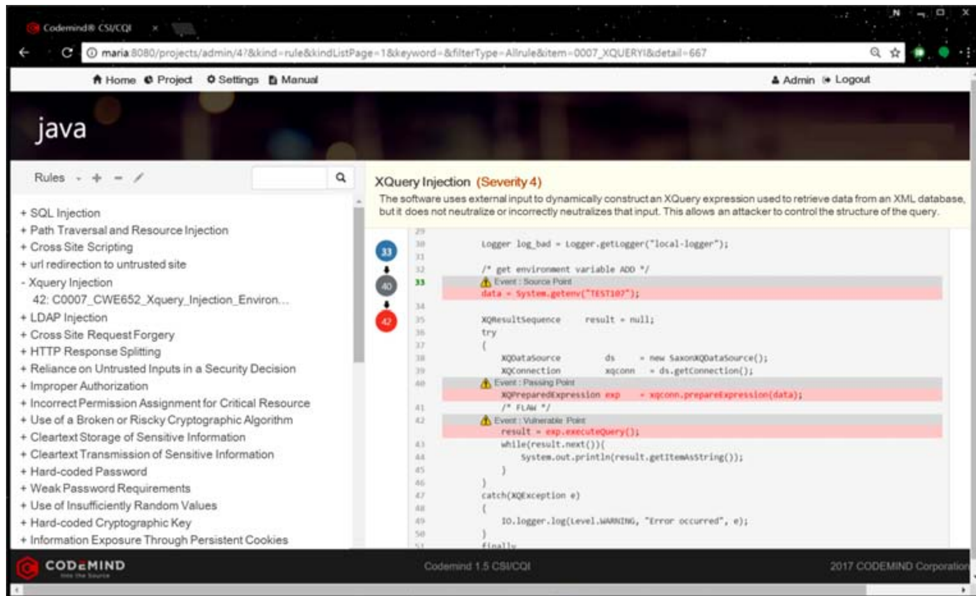

Fig. 1: CODEMIND® CSI/CQI

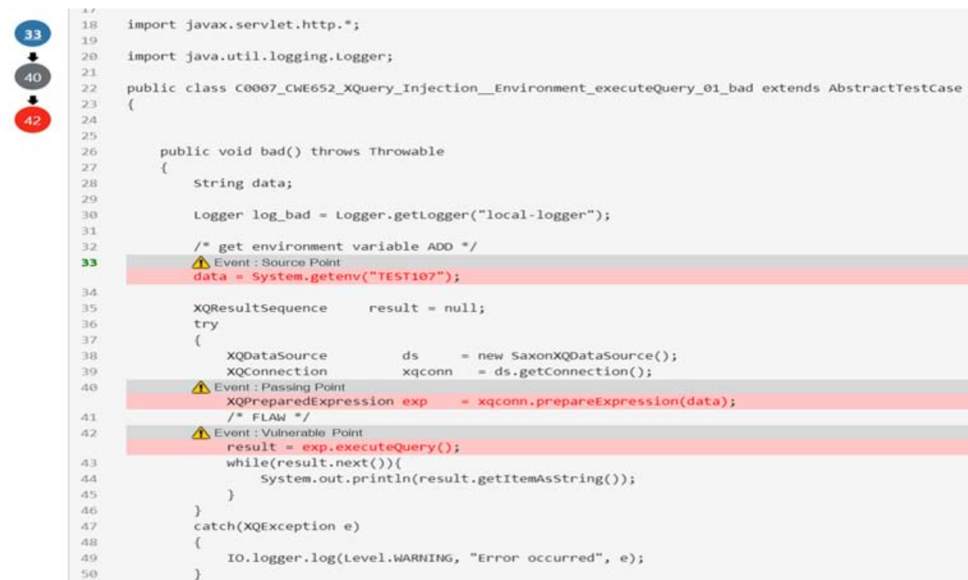

Fig. 2: XQuery injection found by CODEMIND® CSI/CQI

Second, CODEMIND® developer provides the on-the-fly mode in which you can review the recovered warnings even before the analysis finishes. The tool and the user can work simultaneously and the whole amount time can be reduced.

Figure 4 shows the defect tracing for XQuery injection warning. The red colored statement "result = exp.executeQuery()" is the statement in which the warning is found (marked with "!"). You can follow the yellow arrows in reverse to find the orange colored statement that is the origin of the warning "data = System.getenv ("TEST107")". Figure 2 and 4 show the same warning tracing. The difference is that the former gives textual information and the latter gives browsable and graphical information.

Now, it is time to find out the usefulness of these additional features on-the-fly mode and defect tracing graphs of CODEMIND® developer comparing with CODEMIND® CSI/CQI.

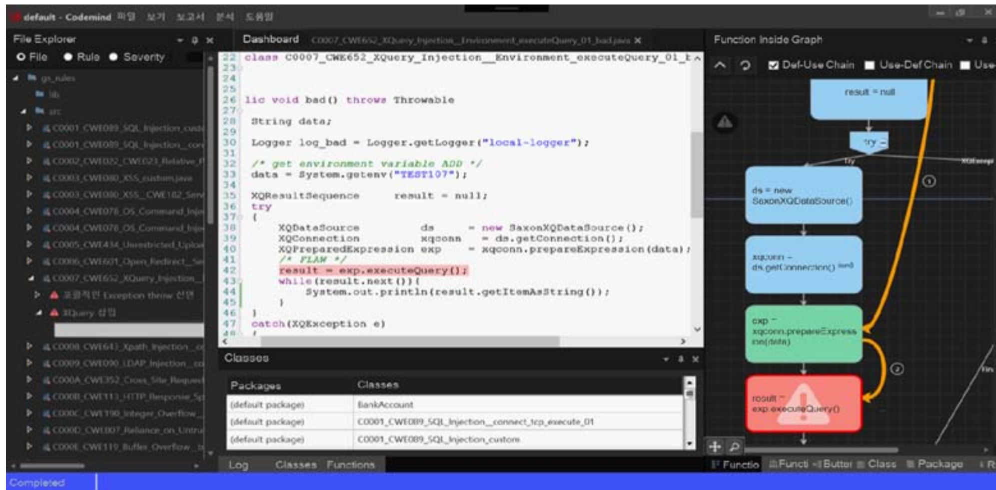

Fig. 3: CODEMIND® developer

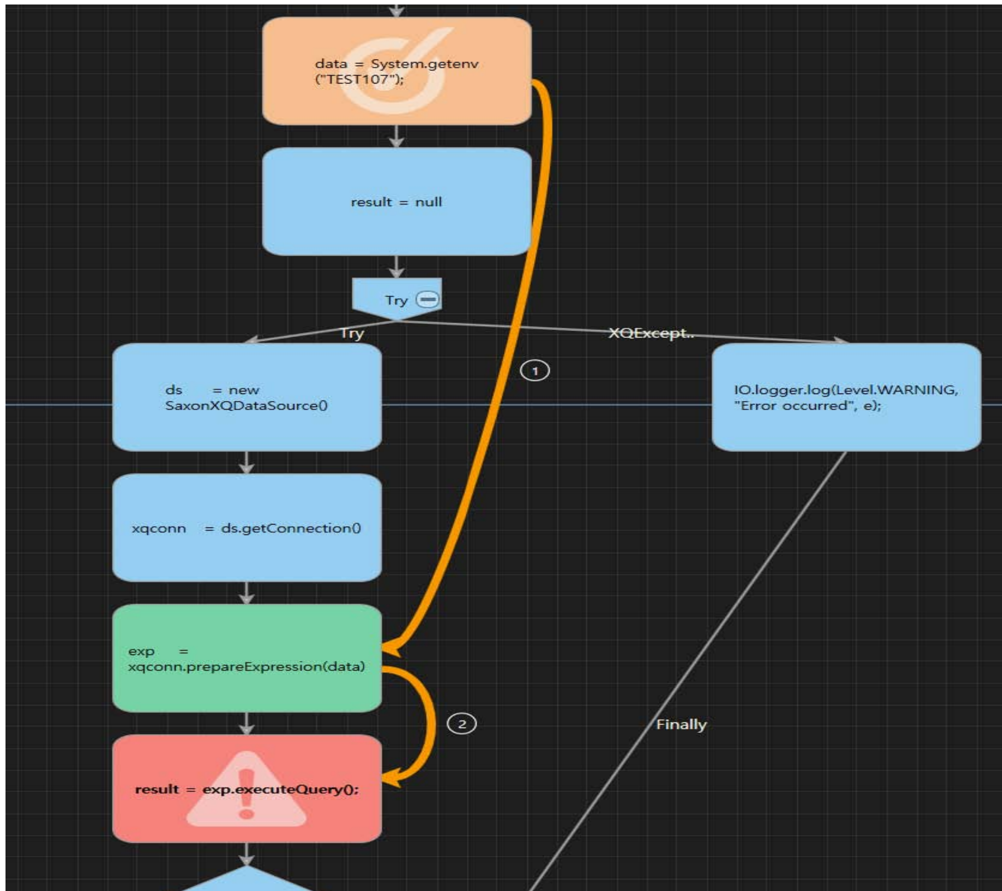

Fig. 4: XQuery injection tracing graph

Comparative usability study of the on-the-fly static analyzer: We conduct a test for testers to detect vulnerabilities of sample programs in three ways: manual vulnerability inspection, vulnerability inspection using a server-based vulnerability analyzer, CODEMIND® CSI/CQI and vulnerability inspection using an on-the-fly

vulnerability analyzer, CODEMIND® developer. The objectives of our experiments can be enumerated as follows:

- Usefulness of automatic analysis tool: to verify the usefulness of using automatic analysis tools in secure coding compared with manual inspection
- Usefulness of on-the-fly style and visualized information; to verify the usefulness of additional features of the on-the-fly analysis tool compared with the server-based analysis tool
- Further, improvements; to derive improvements of our analysis tools from vulnerability inspection experiences of general users

Design of our test: We selected 14 security weaknesses for our test which are SQL injection (CWE-89), path traversal (CWE-22), OS command injection (CWE-78), XPath injection (CWE-643), LDAP injection (CWE-134), Use of hard-coded password (CWE-259), use of hard-coded cryptographic key (CWE-321), information exposure through comments (CWE-396), exceptional iteration (CWE-835), information exposure through an error message (CWE-209), detection of error condition without action (CWE-390) and improper handling of exceptional conditions (CWE-396) (Johnson *et al*., 2013). The weaknesses are selected considering famous weakness lists such as SANS Top 25 (Bradley *et al*., 2012) and severity of related vulnerabilities which are reported in public databases such as NVD (Emanuelsson and Nilsson, 2008; Li and Cui, 2010).

Static vulnerability analyzers inherently have false-negative defects which are actual vulnerabilities not reported and false-positive warnings which are false alarms. Therefore, testers much check whether vulnerabilities reported can be actually exploited by attack. In order to test the productivity features related to checking false alarms, out test suites are composed of programs containing vulnerabilities (bad codes) and programs their vulnerabilities are remediated (good codes). A test set is consisted of 50 programs which were selected from Juliet Test Suite (Johnson *et al*., 2013) or coded manually. Three test sets are constructed and used in our experiment for fairness.

We built a test system which records the progress of inspection for each tester during experiments. For each sample program, we record each time point when testers start inspection and complete finding and fixing a vulnerability.

Eleven subjects participated in our experiment where they had general programming knowledge. Because testers did not have much knowledge of secure coding or security vulnerabilities, we taught them the basic concepts of secure programming, the forms of weaknesses, detection methods and how to use the analysis tools. Also, some preliminary exercises were conducted for manual and tool-based inspections before actual experiments.

Manual program inspection: The manual program test means to examine whether a vulnerability exists in given programs by referring to only the vulnerability-related books and websites without using any automatic analysis tools. We provided testers the software development security guide, the security weakness diagnosis guide and the CWE security weakness information web site (Johnson *et al*., 2013). Before the inspection, we provided testers with the list of 14 weaknesses that could be included in the sample programs and notifies the existence of programs which have no vulnerability.

Four hours were given for testing 50 potentially vulnerable programs. We divided eleven testers into three groups and the three benchmark suites were used by each group for the manual test, the server-based analyzer test and the on-the-fly analyzer test in a non-overlapping manner to minimize the error due to the difference of the difficulty among the benchmarks suites.

Inspection using our server-based analysis tool: Vulnerability testing using a server-based analyzer was performed using CODEMIND® CSI/CQI. CODEMIND® CSI/CQI uploads a Java project file to the server and runs a vulnerability check on this project. The analysis can be done only after all the project codes are provided. As with the other two tests, the analysis time was limited to 4 h.

Inspection using our on-the-fly analysis tool: Vulnerability inspection using an on-the-fly analyzer was performed using CODEMIND® developer to check for vulnerabilities in given Java programs. CODEMIND® developer provides partial analysis results on the fly even before all the analysis finish which can save the overall inspection time. Also, it presents visualized program properties which are related to detected program vulnerabilities which is very helpful to find out false positive warnings. For each vulnerability report, we can make use of visualized program properties related to the vulnerability such as data flow and control flow.

RESULTS AND DISCUSSION

Table 1 shows the summary of our experiment result. We extracted those results from the raw data which records every time points of vulnerability detection and correction for every sample program and every

tester. Each number in the table is the average value from eleven testers and sample programs. The rows of the table show the following results.

Analyzed samples: The number of samples which are examined by testers among 50 sample programs.

Exact analysis: The number sample programs whose vulnerabilities are identified correctly.

Precision: The ratio of exact vulnerability identification over samples analyzed.

Recall ratio: The ratio of exact vulnerability identification over 50 sample programs given.

Corrections: The number of samples whose vulnerabilities are removed correctly.

Corrections ratio: The ratio of samples whose vulnerability were removed correctly over programs given.

Inspection time: The average time consumed for identifying a vulnerability in a sample program.

Correction time: The average time consumed for removing a vulnerability identified. From the experiment results, we can find the following points which support the usefulness of automatic vulnerability analyzers:

Table 1: Experiment results

Variables	Manual	CODEMIND® CSI/CQI	CODEMIND® developer
Analyzed samples	24.09	42	48.09
Exact analysis	15.82	38.73	44.82
Precision (%)	60.69	92.63	93.18
Recall ratio (%)	31.64	77.45	89.64
Corrections	15.09	26.27	29.27
Correction ratio (%)	30.18	52.55	58.55
Inspection time	5:08	1:40	01:11
Correction time	1:57	1:30	01:06

- The ratio of exact analysis increased by 31.94% with the server-based analyzer and by 32.49% with the on-the-fly analyzer
- The recall ratio increased by 45.81% with the server-based analyzer and 58.0% with the on-the-fly analyzer
- The 67.5 and 76.9% of manual inspection time is saved with the server-based analyzer and the on-the-fly analyzer, respectively
- The 23.1 and 43.6% of manual correction time is saved with the server-based analyzer and the on-the-fly analyzer, respectively

In conclusion, it is apparent that automatic analyzers are very useful for vulnerability detection and correction. In case of manual vulnerability inspection, it is not easy to check many types of security weaknesses at the same time. Therefore, as the type of security weakness is more diversified, vulnerability inspection becomes more difficult and the time required for the inspection increase. In the experiment, testers are given the list of 14 potential vulnerabilities in advance which is very helpful for manual analysis. The manual analysis would be, even more difficult if the list had not been given. Because vulnerability inspection is complicated work in general, there exists some gab between testers in productivity. However, it is observed that the use of automatic analyzers can significantly alleviate these interpersonal differences. Table 2 shows the comparison results among testers. In the case of the manual test, the difference in the recall ratio among testers is up to 62% (68 vs. 6%) and the difference in the average inspection time is up to 8 min. When testers use automatic analyzers, the maximum difference in the detection rate was 42% with the server-based analyzer and 26% with the on-the-fly analyzer. The maximum difference in the inspection time also decreased to 99 sec with the server-based analyzer and 91 sec with the on-the-fly analyzer. The Standard

Table 2: Comparison of results among testers

	Manual		CODEMIND® CSI/CQI		CODEMIND® developer	
Testers	Recall ratio (%)	Inspection time	Recall ratio (%)	Inspection time	Recall ratio	Inspection time
1	48.00	2:10	80.00	1:09	96.00	0:52
2	36.00	4:49	92.00	1:05	92.00	1:01
3	50.00	2:03	90.00	0:59	94.00	0:39
4	18.00	8:16	72.00	2:11	88.00	1:52
5	68.00	2:20	74.00	1:03	92.00	0:44
6	6.00	5:14	50.00	2:38	70.00	1:37
7	14.00	5:04	64.00	2:11	90.00	1:31
8	40.00	4:26	86.00	1:32	92.00	0:51
9	6.00	10:04	72.00	1:57	82.00	2:10
10	30.00	5:03	92.00	1:08	96.00	0:58
11	32.00	6:57	80.00	2:26	94.00	0:45
AVG	31.64	5:07	77.45	1:40	89.64	1:10
SD	19.57	2:24	12.93	0:34	7.63	0:30

Deviation (SD) of recall ratio and average inspection time have also decreased with the automatic analyzers.

In order to analyze the effect of the features of analysis tools on the productivity of vulnerability inspection we compared the performance difference between the testings using the server-based analyzer and the on-the-fly analyzer. Because they share same analysis engine, their analysis accuracy is same. But the difference in user experience can result in the difference in productivity.

CODEMIND® CSI/CQI, a server-based tool, works in conjunction with a code version control tool such as SubVersion. It can analyze a program only after all the code has been developed and does not provide partial results during analysis. Also, it does not provide any visualization of program properties related to vulnerabilities. CODEMIND® developer can analyze the code incrementally and provides partial results during analysis in an on-the-fly style. Also, it has a graphical IDE environment which presents vulnerability inspection results and related program properties such as data/control flow information between statements and modules in a visualized form. From Table 1, we can derive the followings comparison results.

- Number of exact vulnerability reports increased by 12.18% with CODEMIND® developer
- Inspection time was reduced by 29% with CODEMIND® developer compared with the server based tool
- Correction time was reduced by 27% with CODEMIND® developer compared with the server based tool

The on-the-fly analyzer results in better productivity in vulnerability detection and correction. According to feedbacks from the testers, it is reported that visualized data-flow information on intra-procedural graph related to vulnerabilities was very helpful for finding and correcting the origin of vulnerabilities. Also, the on-the-fly style was useful for saving inspection time.

In addition to visualized program properties such as data/control flow information, testers testify that automatic provision of guides about weakness forms and remediation methods was very important and helpful for inspection and correction which is provided by both analyzers. Therefore, every tester has agreed that to provide more proper guides and correction recommendations would be very helpful for improving the productivity of vulnerability inspection.

CONCLUSION

In this study, we introduce two forms of static vulnerability analyzers which are the server-based vulnerability analyzer and the on-the-fly analyzer. Compared with the server-based analyzer, the on-the-fly analyzer can detect vulnerabilities in program codes during program development in an incremental way and present partial vulnerability reports during analysis. Also, the on-the-fly analyzer provides a graphical IDE which presents not only vulnerabilities in program codes but also data and control flow information related to the vulnerabilities in visualized forms.

To examine the usability of the tools, we conducted an empirical experiment in which 11 testers analyze 150 sample programs using three methods which are manual inspection and tool based inspection using two vulnerability analyzers. The experiment shows that automatic analysis tools are indispensable for vulnerability detection and removal. Also, the on-the-fly style and the visualized defect trace graphs are proved to be very helpful for productivity. As far as the knowledge, there are no other empirical experiments on the impact of the on-the-fly style and the visualized defect trace graphs on productivity of vulnerability inspection.

RECOMMENDATIONS

As a future study, we consider an empirical experiment about the usability of other additional features for productivity such as automatic fixation, correct code recommendation, coding habit mining, etc. which will be helpful for building more intelligent static vulnerability analyzers.

REFERENCES

Bradley, M., F. Cassez, A. Fehnker, T.G. Wilson and R. Huuck, 2012. High performance static analysis for industry. Electron. Notes Theoretical Comput. Sci., 289: 3-14.

CWE, 2013. 2011 CWE/SANS top 25 most dangerous software errors. Document Version: 1.0.3, Common Weakness Enumeration (CWE), September 13, 2011. http://cwe.mitre.org/top25/archive/2011/2011_cwe_sans_top25.pdf.

Charest, T., N. Rodgers and Y. Wu, 2016. Comparison of static analysis tools for java using the Juliet test suite. Proceedings of the 11th International Conference on Cyber Warfare and Security, March 17-18, 2016, Boston University, Boston, USA., pp: 431-438.

Chess, B. and G. McGraw, 2004. Static analysis for security. IEEE. Security Privacy, 2: 76-79.

Diaz, G. and J.R. Bermejo, 2013. Static analysis of source code security: Assessment of tools against SAMATE tests. Inf. Software Technol., 55: 1462-1476.

Emanuelsson, P. and U. Nilsson, 2008. A comparative study of industrial static analysis tools. Electron. Notes Theor. Comput. Sci., 217: 5-21.

Fu, X., X. Lu, P. Verger, B.S. Chen, K. Qian and L. Tao, 2007. A static analysis framework for detecting SQL injection vulnerabilities. Proceedings of the IEEE Annual International Computer Software and Application Conference, Volume 1, July 24-27, 2007, Beijing, pp: 87-96.

Johnson, B., Y. Song, E. Murphy-Hill and R. Bowdidge, 2013. Why don't software developers use static analysis tools to find bugs?. Proceedings of the 35th International Conference on Software Engineering (ICSE), May 18-26, 2013, IEEE, San Francisco, California, ISBN:978-1-4673-3073-2, pp: 672-681.

Li, P. and B. Cui, 2010. A comparative study on software vulnerability static analysis techniques and tools. Proceedings of the IEEE International Conference on Information Theory and Information Security (ICITIS), December 17-19, 2010, IEEE, Beijing, China, ISBN:978-1-4244-6942-0, pp: 521-524.

Mantere, M., I. Uusitalo and J. Roning, 2009. Comparison of static code analysis tools. Proceedings of the 3rd International Conference on Emerging Security Information, Systems and Technologies SECURWARE'09, June 18-23, 2009, IEEE, Athens, Glyfada, ISBN:978-0-7695-3668-2, pp: 15-22.

Ramos, A., 2016. Evaluating the ability of static code analysis tools to detect injection vulnerabilities. Ph.D Thesis, UMEA University, Umea, Sweden.

Schiela, R., 2017. SEI CERT coding standards. BSc Thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania.

Srinivasan, N. and P. Thambidurai, 2007. On the problems and solutions of static analysis for software testing. Asia J. Inform. Technol., 6: 258-262.