International Journal of Advances in Software Engineering & Research Methodology– IJSERM
Volume 2: Issue 2 [ISSN : 2374-1619]
Publication Date : 30 October, 2015

Reducing False Alarms in Static Code Analysis with Test Code Mutants

Hyun Woo Park and Kyung-Goo Doh

***Abstract*— Software is often exposed to safety accidents due to hacking and defects. Most of the accidents are caused by bugs and security vulnerabilities in source code. The bugs and vulnerabilities should be eliminated during the development phase before software release. Nowadays, many software developers use static code analysis tools for secure software development. Thus it is necessary to have an effective way of evaluating the quality of static analysis. Despite the advantages of static code analysis, the developers avoid to use it because of the immoderate false alarms. Unless static analysis tool is tested appropriately, the false-alarm rate may be increased. In this paper, we propose a method of automatically generating test codes based on mutation testing techniques.**

Keywords—static code analysis, test code, mutation testing, false alarm, secure coding.

I. Introduction

Trivial and minor software defects delay the delivery of services and cause the loss of money. Especially, software installed in automobiles, aircrafts, and rockets are expensive and safety critical. The defects of their software can even threaten human life. Researchers have proposed various methods that apply test and analysis techniques to identify defects residing in software.

Program analysis is the process of automatically or manually analyzing the behavior of computer programs. There are basically two types of program analysis: static and dynamic analysis. These analyses detect defects and security vulnerabilities in the program. Dynamic analysis is popular and is performed by executing programs. However, it does not guarantee the absence of vulnerabilities or defects. There are many types of dynamic-analysis techniques, such as testing, monitoring, debugging and program slicing [2].

Hyun Woo Park / Dept. of Computer Science and Engineering
Hanyang University ERICA
55 Hanyangdaehak-ro, Sangnok, Ansan, Gyeonggi, Korea

Kyung-Goo Doh / Dept. of Computer Science and Engineering
Hanyang University ERICA
55 Hanyangdaehak-ro, Sangnok, Ansan, Gyeonggi, Korea

Corresponding author

This research was supported by the MSIP(Ministry of Science, ICT and Future Planning), Korea, under the ICT/SW Creative Research program (NIPA-2014-H0502-14-3022) supervised by the NIPA(National IT Industry Promotion Agency)

On the other hand, static analysis is performed without actually executing programs. It can discover vulnerabilities during the development phase, and cover every execution path. Hence, it guarantees the absence of vulnerabilities and defects. These vulnerabilities and defects are easier to correct than the ones found during the testing phase since static analysis finds the root of the vulnerability and defect. There are many types of static-analysis techniques, such as type inference, control-flow and data-flow analysis and model checking [2].

Immature analysis tools often produce false positive results where the tool reports a possible defect and vulnerability that in fact is not. The use of immature analysis tools can also result in false negative results where the tool misses defects or vulnerabilities. In order to properly evaluate analysis capabilities of the static analysis tools, well-selected test codes containing probable defects and vulnerabilities need to be prepared. Incomplete set of test codes might have defects in analysis tools go undetected. Thus the preparation of test-code set is an important factor for an analysis tool.

The practicality and effectiveness of static-analysis tools can be maintained by keeping the balance between false positive and negative rates. It is not simple to reduce both false positive and negative rates at the same time. In order to minimize the false-negative problem, analysis tool has to consider relations between function calls and data sensitively. However, it increases the false positive rate as well. False positive and negatives are in inseparable relation.

Immoderate false-positive problem makes software developers frustrated. Manual reviewing or auditing to check the authenticity of analysis results is very time-consuming. Furthermore, high false-positive rates have developers lose their trust in analysis tools and regard actual defects and vulnerabilities as false.

Hence it is necessary to minimize inseparable relation between false-positive and false-negative problem. Despite of the advantages of static analysis tools, the developers avoid using them due to immoderate false alarms [1]. Moreover, preparing test codes for the analysis tools requires a great deal of efforts. To guarantee the quality of static-analysis tools, it is necessary to properly prepare effective and abundant test codes. In this paper, we study and suggest a method of generating further effective and abundant test codes by applying mutation-testing techniques to initial static-analysis results.

II. Types of Test Codes

Two types of test codes are used to evaluate the analysis precision of static analysis tools. The one type is the sample codes publicly available in various sites defining and explaining defects and vulnerabilities [3]. The other type is a prepared test code created manually by analysis-tool developers [4, 5]. Using these two types as test codes has advantages and disadvantages.

CVE (Common Vulnerabilities and Exposures) [7] and US-CERT (United States Computer Emergency Readiness Team) [8] have been established to provide a unified, measurable set of software weaknesses, vulnerabilities and secure-coding guidelines. Documents of CVE and US-CERT available publicly provide sample codes that show the existence and nonexistence of defects or vulnerabilities. Tool developers could use these sample codes as test codes. They might not be sufficient, however, because they are only examples and not complete as test code. Manually constructing test codes based on the samples code is very time-consuming.

Some test codes can be prepared readily by tool developers. However, developers tend to write the test codes that work only for the specific logic implemented in their tool, missing some vulnerabilities and defects. Thus the test code could be written in such a way that it only works in very simple and limited way [5]. This may increase false-alarm rates of the static-analysis tool.

Two types of test codes used to evaluate analysis capabilities of the tool have obvious advantages and disadvantages. However, it is wise to use sample source codes from CVE and US-CERT because they tend to be more reliable. Therefore, an approach which starts from sample source codes provided and makes them more complete as test codes is in need.

III. Related Research

Researches have been done to compensate negative aspects of static analysis. Recent advances in static-analysis technologies have brought forward tools that do deeper analyses that find more defects and vulnerabilities, and produce a reasonable amount of false alarms [1]. Due to the considerable high cost of reviewing and auditing false alarms, extensive researches are still ongoing.

Several static-analysis tools have been developed through the years in order to discover defects and vulnerabilities in the software development phase. Research on evaluating five modern static analysis tools (ARCHER [9], BOON [10], Poly-Space C Verifier [11], Splint [12], and UNO [13]) using open-source code examples containing 14 exploitable buffer overflows has been conducted [6]. The research showed that only Poly-Space and Splint have reasonable detection rates. Poly-Space and Splint had average detection rates of 87% and 57%, respectively. However, the average false-alarm rate of these two tools is still about 50%. It has both high detection and false-alarm rates. High false-alarm rates makes software developers review and audit source code manually which is painful.

Mutation testing is fault-based test method, and it is used to design additional software test set and evaluate the quality of existing software test set. It identifies the location of defects and vulnerabilities in source code. Faults are introduced into the program source code by creating a set of faulty versions, called mutants [14]. These mutants are created from the original program source code by applying a mutation operators which introduce a single syntactic change or fault to source code. The test sets are used to execute the created mutants with the goal of causing each mutant to fail the test set. If a test set cannot distinguish a mutant, it requires additional test cases to distinguish all the mutants. It improves the quality of the test set. Mutation operators are various, Table 1 shows basic ten mutation operators.

Table 1. Mutation Operator

Operator	Description
ABS	Absolute value insertion
AOR	Arithmetic operator replacement
LOR	Logical operator replacement
ROR	Relational operator replacement
UOI	Unary operator insertion
UOD	Unary operator deletion
COR	Conditional operator replacement
SOR	Shift operator replacement
ASR	Assignment operator replacement
SVR	Scalar variable replacement

The number of mutants which can be generated increases exponentially. Thus, applying the mutation operator to every location of source code is very expensive and time-consuming. Since mutation testing execution cost is considerably high, researchers have proposed a selective mutation technique, which uses a subset of the mutation operators instead of using all operators [15]. Even with the selective mutation technique, the cost of creating and testing mutants is still not reasonable. Hence, extensive researches about the reduction of mutation testing cost are still ongoing.

IV. Generating Test Code Mutants

CVE and US-CERT provide source-code examples that show security weaknesses and vulnerabilities, as well as cases that violate secure-coding guidelines. Figure 1 shows a C program example given in US-CERT documents. This particular secure-code guideline numbered FLP30-C says, "Do not use floating-point variables as a loop counter"[8]. Floating-point numbers represent real numbers. It is often mistakenly assumed that they can represent any simple fraction exactly. Floating-point numbers are subject to representational limitations unlike an integer number, and binary floating-point numbers cannot represent all real numbers exactly. In addition, because floating-point numbers are able to represent large values, it is often mistakenly assumed that they can represent all significant digits of those values as well. The first for-loop in Figure 1 uses variable of type float as a loop counter, it has the problem that there is a possibility of the inaccurate number of iterations. This loop may iterate either nine or ten times. However, the second for-loop has no problem since this for-loop uses integer type variable as a loop counter. It iterates exactly ten times.

```
for (float x = 0.1; x <= 1.0; x += 0.1 ){
    /* Loop may iterate 9 or 10 times */
}
for (int count =1; count <= 10; count +=1 ){
    /* Loop iterates exactly 10 times */
}
```

Figure 2. CERT FLP30-C example code

However, these test cases are not sufficient since there are other similar types such as double and long double. If the test cases in Figure 1 alone are used to evaluate the precision of static analysis tool, false-negative rates may rise because some of the necessary test cases were omitted. In contrast, test cases that do not include other types such as integer, short, long, etc. may raise false-positive rates.

Testing analysis tool with just a few representative examples might be insufficient and might not cover every possible case. Defects and vulnerabilities other than what is on the published documents ought to be additionally reviewed. Creating loop code with all types as a loop counter variable in manual requires a great deal of efforts and it is very time-consuming. Furthermore, manual construction of test codes is always prone to have missing cases. Developers could also make mistakes.

Most analysis tools are able to locate the exact location in the source code where defects or vulnerabilities occur. Applying each mutation operator to every location of original test code is inefficient. It creates a lot of useless test-code mutants [15]. By using only the reported location normally represented as a line number and column number range, appropriate and sufficient test-code mutants can be generated in a reasonable time.

The overview of how appropriate and sufficient test-code mutants are generated is shown in Figure 2. The test code is parsed to create AST(Abstract Syntax Tree). The static analysis tool analyzes the AST. The analysis result is the set of location information where defects or vulnerabilities reside.

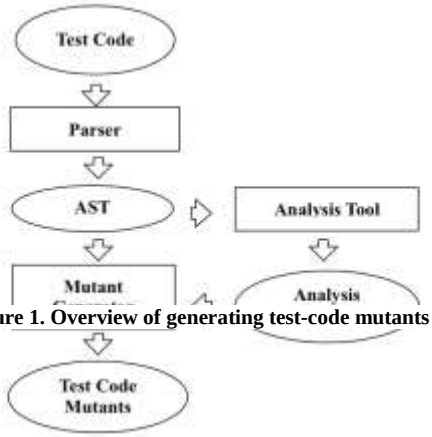

Figure 1. Overview of generating test-code mutants

Mutant generator takes the AST and the location as input, and then it generates the several test-code mutant ASTs. The generated mutant ASTs are converted to source code. Test-code mutants are adequate because they are modified from the analysis result. In addition, since it applies a mutation operator to the specific location of source code, it is not very time-consuming to generate. Figure 3 shows an example of the generated test-code mutants.

The analysis tool which is developed with the above original test-code reports the floating-point number type variable. Test code mutants are generated with line information and original test code. Test code mutants with all types are generated, and they are analyzed again. An appropriate analysis capability testing reduces the false alarm

```
for (float x = 0.1; x <= 1.0; x += 0.1 ){
    /* Loop may iterate 9 or 10 times */
}
```

Original Test Code

```
for (float x = 0.1; x <= 1.0; x += 0.1 ){
    /* Loop may iterate 9 or 10 times */
}
for (double x = 0.1; x <= 1.0; x += 0.1 ){
    /* Loop may iterate 9 or 10 times */
}
for (int x = 0.1; x <= 1.0; x += 0.1 ){
    /* Loop may iterate 9 or 10 times */
}
. . . . . .
```

Test Code Mutants

Figure 3. Example of generated test-code mutants

rates.

V.Conclusion and Future works

The history of static analysis is shorter than that of dynamic analysis. However, both analysis techniques have clear advantages and disadvantages. Ironically, the best way to certify that the source code has the least amount of defects and vulnerabilities is by combining both the static and dynamic analysis. The commercial use of static analysis is growing and extensive researches on compensating disadvantages of static analysis have been carrying out. Static analysis discovers defects and vulnerabilities of software during the development phase, and it examines all possible execution paths and variable values, so it does guarantee the absence of vulnerabilities and defects. It also discovers the root of the vulnerability and defect, so it is easier to correct them.

However, the false-alarm problem of static code analysis is inevitable. Despite its advantages, software developers avoid using it because of the immoderate false alarms. Researches on comparing detection and false-alarm rates of several commercial static-analysis tools with open source have been done [1, 6]. It is impossible to eliminate all false alarms, but it is possible to minimize the false alarm rates.

Effective and abundant test-code generation lets us not only see false-positive rates to be decreased, but also false-negative rates. TO accomplish that, various and effective mutation operators of mutation testing can be applied to static analysis. Both mutation testing and static code analysis discover defects and vulnerabilities of the software. However, static analysis can be performed faster than mutation testing. Test code mutants can be generated programming-language independently in a reasonable time. The reason is that it requires only the source code and the location where the defect or vulnerability exists. Preparing abundant and effective test codes is the key to solving the false-alarm problem.

As future works, we plan to generate more effective and adequate test codes. There are various kinds of static analysis [2] such as control-flow analysis, data-flow analysis, model checking, etc. In this paper, we only consider type mutation techniques. However, flows between functions and data can also be mutated with other mutation operators such as statement mutation and value mutation. We would like to create more test-code mutants to reduce false-alarm rates of static analysis.

References

1. I. Gomez, P. Morgado, T. Gomes, and R. Moreira, “An overview on the Static Code Analysis approach in Software Development,” Tech. rep., Faculdade de Engenharia da Universidad do Porto 2009
2. Program Analysis, http://en.wikipedia.org/wiki/Program_analysis
3. R. K. McLean, “Comparing static security analysis tools using open source Software,” 6th IEEE Int. Conf. SW Security Reliability Companion (SERE-C), pp. 68-74, Gaithersburg, U.S.A., Jun. 2012
4. T. Hofer, “Evaluation static source code analysis tools,” M.S Thesis, School Compt. Common. Sci. École Polytechnique Fédérale de Lausanne, Mar. 2010
5. J. Bang, and R. Ha, “Validation Test Codes Development of Static Analysis Tool for Secure Software,” J. KICS, vol. 38C, no.05, pp. 420-427, Oct. 2013
6. M. Zitser, R. Lippmann, and T.Leek, “Testing static analysis tools using exploitable buffer overflows from open source code,” 12th ACM SIGSOFT Int. Proc. symposium on Foundations of Software engineering, vol. 29, no.06, pp. 97-106, Nov. 2004
7. MITRE, Common Vulnerabilities and Exposures, Retrieved Dec, 18, 2014 from https://cve.mitre.org
8. US-CERT Vulnerability, US Computer Emergency Readiness Team, Retrieved Dec, 30, 2014 from http://cert.org
9. Y. Xie, A. Chou, and D. Engler. “Archer: Using symbolic, path-sensitive analysis to detect memory access errors,” 11th ACM SIGSOFT Int. Proc. Symposium on Foundations of Software engineering, vol. 28, no.05, pp. 327-336. 2003
10. D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken, “A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities,” In NDSS, pp. 3-17. Feb. 2000
11. PolySpace C Verifier, http://www.polyspace.com/download.htm
12. D. Evans and D. Larochelle, “Improving security using extensible lightweight static analysis,” IEEE, Software, vol 19, no 01, pp. 42-51. 2002
13. G. J. Holzmann, “Static source code checking for user-defined properties,” In Proc. International Conference on Integrated Design and Process Technology, June. 2002
14. S. Kim, J. A. Clark, and J. A. McDermid, “Class Mutation: Mutation Testing for Object-Oriented Programs”, Proc. ObjectDays Conference on Object-Oriented Software Systems, Oct. 2000
15. A. Jefferson, Gregg Rothermel, and Christian Zapf. "An experimental evaluation of selective mutation." Proceedings of the 15th international conference on Software Engineering. IEEE Computer Society Press, 1993

About Author (s):

Hyun Woo Park is a Master’s Student in the Department of Computer Science and Engineering at Hanyang University, Republic of Korea. He received his B.S. in computer science at the University of Waikato, New Zealand. His research interests are in the area of programming languages and software engineering.

Kyung-Goo Doh is Professor in the Department of Computer Science and Engineering at Hanyang University ERICA, Republic of Korea. He received his Ph.D. in computer science at Kansas State University. His research interests are in the area of programming languages, program analysis, software security, and software engineering.