Wireless Pers Commun (2016) 89:777–793
DOI 10.1007/s11277-015-3152-1

Software Vulnerability Detection Methodology Combined with Static and Dynamic Analysis

Seokmo Kim[1] · R. Young Chul Kim[2] · Young B. Park[3]

Published online: 17 December 2015

Abstract Software vulnerability is the attack surface. Therefore, vulnerabilities innate in software should be detected for software security assurance. Vulnerability detection method can be divided into static vulnerability detection and dynamic vulnerability detection. Static vulnerability detection is more commonly used for vulnerability detection. This method has many benefits, but it also creates false positives. Therefore, this paper proposes a method to combine static and dynamic detection to reduce false positives created from static vulnerability detection. The proposed method verifies the vulnerability by implanting a fault, based on the information received from static code analysis.

Keywords Vulnerability · Instrumentation · Fault injection · Model-to-text transformations

Special Issue: "Convergence Interaction for Communication", Guest Edited by Prof. Jong Kyung Ryu, jkryu.hci@gmail.com.

✉ Young B. Park
ybpark@dankook.ac.kr

Seokmo Kim
seokm0@naver.com

R. Young Chul Kim
bob@selab.hongik.ac.kr

[1] Department of Computer Science & Engineering, Dankook University, Yongin, Republic of Korea

[2] Department of Computer Information Communication, Hongik University, Sejong, Republic of Korea

[3] Department of Computer Science, Dankook University, 119, Dandae-ro, Dongnam-gu, Cheonan-si 31116, Chungnam, Republic of Korea

1 Introduction

As software becomes larger and complex, vulnerability is also increasing. According to the statistics of National Vulnerability Database [1] (Fig. 1), 19 new vulnerabilities in average were reported daily in 2014. This is notably higher than the statistics of 2013. In addition, approximately 80 % of the newly reported vulnerabilities are from the application layer. The fact that the attack on the application layer increased can be seen on the report of SANS in 2009 [2]. Therefore, to decrease these risks and guarantee software security, analysis of innate vulnerability in software is needed [3]. Due to remarkable growth of mobile devices [4], mobile application got especially bigger and complex. Therefore, the threat to the software used in mobile devices is increasing consistently. Also, as mobile devices are always connected to the network, it is easy for attackers outside to access it [5]. In addition, as users carry their mobile devices persistently, it holds much personal information [6]. For example, it may hold the user's current location, contacts, or even medical information. Therefore, to decrease the security threat in the mobile application field, the effort to detect software vulnerability is needed.

Software analysis method such as the vulnerability detection is largely divided into static analysis method and dynamic analysis method. Static analysis inspects the code itself without operation, and dynamic analysis inspects the behaviors during the runtime. Static analysis is mostly used when detecting vulnerabilities, but due to low accuracy, it produces many false positives [7]. Many false positives of static analysis have the advantage of showing every vulnerability issue that has even the slightest possibility to the testers, and making them to check it. However, too many false positives can make the user stop using the method (or tool), or waste too much time checking it [8]. So there needs to be a method to verify the vulnerabilities detected by static code analysis.

Therefore, this paper proposes a way to verify the vulnerability detected from static code analysis which is actually inherent in software, and exploitable for an attack. The proposed method is a combined method of both static and dynamic analysis. This method detects all possible vulnerabilities through static analysis. Then, the vulnerabilities should be verified by dynamic analysis. Therefore, dynamic analysis should use the information acquired from static analysis. However, as the vulnerability information acquired by static analysis is in its code level, it is hard to be applied to dynamic analysis which cannot be operated in the code level. To solve this problem, this paper uses instrumentation. And to automatically create instrumentation code, it used model-to-text transformations.

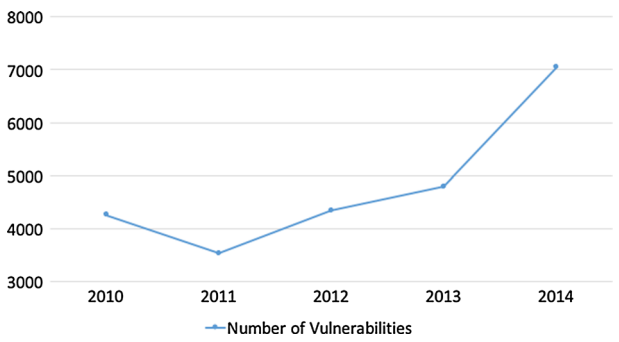

Fig. 1 The number of vulnerabilities from National Vulnerability Database

The proposed method has the high coverage, short analyzing time, and various vulnerability processing capacity of static analysis, and also the high accuracy of dynamic analysis. In addition, this method can check whether the attacker can access the vulnerability through vulnerability-centric data flow analysis.

This paper uses instrumental techniques that automatically insert specific code to the source files under analysis to materialize a tool to combine static analysis and dynamic analysis to verify the proposed idea. Also, an experiment to analyze SQL injection vulnerability was done by analyzing sample programs with the materialized tool.

The rest of this paper is organized as follows. Sections 2 and 3 provide a brief description of existing techniques for vulnerability detection, Sect. 4 describes instrumentation technique, Sect. 5 describes model-to-text transformations. In Sect. 6, we present our proposal. Then, in Sect. 7, we describe a prototype implementation of out approach and the results of its experimentation. Finally, conclusions are given in Sect. 8.

2 Static Vulnerability Detection

Static vulnerability detection is a way of analyzing the source code (or the binary code or the object code) without actually executing programs. The advantage of static vulnerability detection is that is has 100 % code coverage, and that it can deal with various vulnerabilities. And also, as it is done without the execution of programs, the analyzing time is short and it is suitable to be applied to early Software Development Life Cycle. In addition, as the analysis is done at the code level, it is easy to edit the source code to remove the vulnerability. However as there is no information of behavior of the program, every vulnerability with even the slightest possibility is reported, and many false positives are included in the detected vulnerabilities. The techniques that can be used in static vulnerability detection include pattern matching, lexical analysis, parsing, data flow analysis, taint analysis.

2.1 Pattern Matching

Pattern matching is finding the string that concords with the vulnerability pattern set beforehand from the source code. Finding the function such as strcpy() that can cause buffer overflow from C-language is an example. These pattern matching has the advantage that it can be materialized simply. However, this method produces many false positives. This is because the analysis through simple string matching has no information about the structure or meaning of the program. Flawfinder [9] is the tool in which this method is applied.

2.2 Lexical Analysis

Lexical analysis alters the code into token stream, and vulnerability pattern matching is operated using this token stream, Token stream makes pattern matching more accurate. This is because the lexer can manage the irregular blanks and code form. This technique is very simple and fast. But this technique also cannot decipher the structure and meaning of the program. Therefore, it still causes many false positives. ITS4 [10] is a tool using this technique.

2.3 Parsing

Parsing parses the source code, and expresses the source code into abstract syntax tree. This parsing tree analyzes the program's structure and meaning. Lexical analysis cannot separate the function and variable with the same name. However, AST(Abstract Syntax Tree) analysis can decipher the types of identifiers. So, pattern matching using AST is complex, but quite accurate. So it can detect vulnerabilities that cannot be found using lexical analysis. PMD [11] is a tool with this technique applied.

2.4 Data Flow Analysis

Data flow analysis is a traditional compiler technique to solve buffer overflow or format string problem. This technique can also be used in vulnerability detection. This technique is a technique that collects possible expression or variable cost while the program is operating.

3 Dynamic Vulnerability Detection

Dynamic vulnerability detection analyzes vulnerability based on the behavior of software. Therefore, the actual execution of a program is done. These techniques are operated as if a malicious user attacks software. So the vulnerabilities detected from this analysis have less false positives because of its high accuracy. However, this technique has a drawback of restricted code coverage. In addition, it has a difficulty of creating many test cases, and it uses much time to detect vulnerability. Techniques that can be used in dynamic vulnerability detection are such as fault injection, fuzz testing.

3.1 Fault Injection

Fault infection is a technique that injects various errors in a system to see how the software reacts to the error. Fault injection is a good way to test the tolerance or the robustness of a system against errors, and it is especially suitable for stress test in which critical error that does not appear in typical test is injected [12]. This testing technique makes the system enter an error, and analysis the reaction against the error or fault [13]. The objects that can be injected are such as protocol, register, memory value, or code.

Fault injection is comprised of Hardware Implemented Fault Injection(HWIFI) and Software Implemented Fault Injection(SWIFI). SWIFI has two methods of compile-time injection and runtime injection. Compile-time injection adjusts the source code in order to inject an error in the system. This method may edit the existing source code, but mostly it injects an error-causing code without adjusting the existing source code. And runtime injection uses a software trigger to inject an error while the software is operating. There are many fault injection techniques as explained, and the mostly used way to test high level software is instrumentation [14]. This paper also uses instrumentation to inject errors. Instrumentation is explained in Sect. 4 below.

3.2 Fuzz Testing

Fuzz testing provides random data from the input value to test if the program is accurately working. This method is easier to materialize than fault injection. This is because it is

simple to design the test, and no advance information is needed. However, this method has the limit that it can be done only at the entry point of the program. In addition, as random entry is done, there are too many test cases. Therefore, the time spent on analysis is too much. Web scanner is one of these tools.

4 Instrumentation

There are many techniques to collect a program's runtime information [15]. In general, runtime data collection method can be divided into hardware-assisted and software-only collection scheme. Hardware-assisted collection scheme requires a hardware device that is added to the system for data collection. So it is costly. On the other hand, software-only collection schemes are less costly because it does not require an additional hardware device. There are two approaches of software-only collection schemes. One is simulating, emulating, or translating a program's code, and the other is instrumenting a program's code. Code emulation technique simulates the hardware execution of the target program by fetching, decoding, and emulating each instruction of the test program. The main advantage of this tool is that it supports cross-simulation, and that the code can be operated without hardware. This technique, compared to instrumentation technique, is slower than instrumented binary when collecting runtime information such as emulated binary.

Instrumentation technique operates by rewriting the target program while the program is operating to collect runtime information. Logical behavior of the instrumented target program is alike the one before instrumentation. Then the native hardware of the original program still operates the program. But the routines for data collection are invoked at the righteous point during the operation of target program to collect runtime information. The mechanisms to invoke run-time data collection routines consist of microcode instrumentation, operating system trapping, and code instrumentation. The most typically used approach is to directly edit the program's code. This approach injects extra instructions to the target program to collect wanted runtime information. The data collection of this method causes minimum overhead. This is because the programs in this method run in native mode. In this, only the overhead of procedure call to invoke data collection routine may exist.

Code instrumentation can be divided in three due to the point in which the code is instrumented. Executable instrumentation (Late code instrumentation) is done after the executable is created. This does not require source file. However, as there is less information that can be used in binary file this method is most complex, and the instrumented binary undergoes performance and liability problem. Next, link-time instrumentation is done while object linking. This method maintains source code independence, and it is easier to instrument a program. Lastly source-level instrumentation is done before compiling. In this method, it has simple code instrumentation process because the code information is enough. And it is able to make complex traces. This method requires a program's source code.

Also instrumentation can be used variously according to the inserted code. For example, it can be used as code tracing to get runtime information, debugging to find the programming error in the software development stage, profiling to measure dynamic program's movement, performance counter to appraise performance, computer data logging to trace an application's event [16]. Tools that can use instrumentation are Inter Pin framework [17], DynamoRio [18].

5 Model to Text Transformations

Traditionally models have been used in software development to define and understand the problem domain or the different aspects of a system's architecture. Especially, modeling in model-driven architecture is the most important point in software development stage. Reusing model and creating a code through the model is increasing productivity. Model to model transformation is changing Platform Independent Model(PIM) into Platform Specific Model(PSM), PIM to PIM, or PSM into PSM [19]. On the other hand, model-to-text transformation is changing the model into text product such as code, specifications, or documents. These transformations need standard. For example, the statement of Object Management Group [20] for model transformation language is one of these standards. QVT standard of OMG standard is a standard for transformation, and MOF Model to Text standard is a standard for model-to-text transformation.

6 Hybrid Vulnerability Detection

Vulnerability detection using static code analysis has high code coverage, can handle various vulnerabilities, and has an advantage of short time of analyzing, but it has a drawback of producing many false positives. The reason for the production of false positives is lack of analysis data of static code analysis. The analysis data in static code analysis is only the code. The code has the program's static information. Therefore, static code analysis does not include the program's dynamic(runtime) information. This limit makes certain static code analysis method to only use the code without execution to presume the program's behavior. However, this presumption is not accurate. Therefore, if this approach is used in vulnerability detection, it produces a lot of false positives. In addition, vulnerability detection needs clear specification of the vulnerability. This specification is defined based on the syntax and semantics of vulnerability code. Therefore, the static pattern is easy to be expressed in this specification, but the dynamic pattern is hard to be expressed. So, clear specification of vulnerabilities requiring a program's dynamic behavior information is impossible. As a result, due to unclear specification, vulnerability detection through static code analysis creates false positive.

Therefore, a process to judge whether the vulnerability detected through static security analysis really exists in the software is needed. Many false positives have the advantage of making testers or analyzers to check every possibility. However, as many false positives become a reason for testers or analyzers to completely stop using the method (or tool), it is a problem.

Therefore, the way to identify the vulnerability (false positive) among the vulnerabilities detected through static code analysis that really exists in the software and can be used in an attack is needed. And this identifying cannot be done with only static analysis. So the approach of combining static analysis and dynamic analysis is needed. However, as static code analysis is in the code level, the analysis result is also code level information. On the other hand, code level information of static code analysis cannot be used in dynamic analysis because dynamic analysis is not in the code level. Therefore, to combine static analysis and dynamic analysis, a method to use the code level information of static analysis in the dynamic analysis should be proposed. This paper therefore uses instrumentation technique to solve this problem.

Briefly explaining the method this paper proposes, firstly vulnerability is detected through static code analysis. Then, the code to verify these detected vulnerabilities is inserted into the target program's source code. Then by actually executing the program with the verification code, it is analyzed whether the detected vulnerability can be used in a real attack. So, the proposed method comprises of vulnerability detection stage through static code analysis, instrumentation stage to link static code analysis and dynamic analysis, dynamic analysis stage of injecting an error and monitoring the impact on a program. In the first stage of static code analysis, vulnerability is detected through pattern matching using abstract syntax tree. Then the accessibility of the vulnerability is checked through data flow analysis. Next, in instrumentation stage, which is the second stage, instrumentation code is created using the vulnerability information acquired through static code analysis to verify the detected vulnerability through static code analysis. And this code is inserted into the analysis target program. The role of instrumentation code is to inject the error that activates the vulnerability, and then to monitor the effect of the error. The third stage, which is the dynamic analysis stage, actually executes the manipulated program with the instrumentation code inserted. During the runtime, the inserted instrumentation code injects an error, and the activation of the vulnerability is monitored. And the effect of this activated vulnerability is traced. The Fig. 2 below expresses the proposed method.

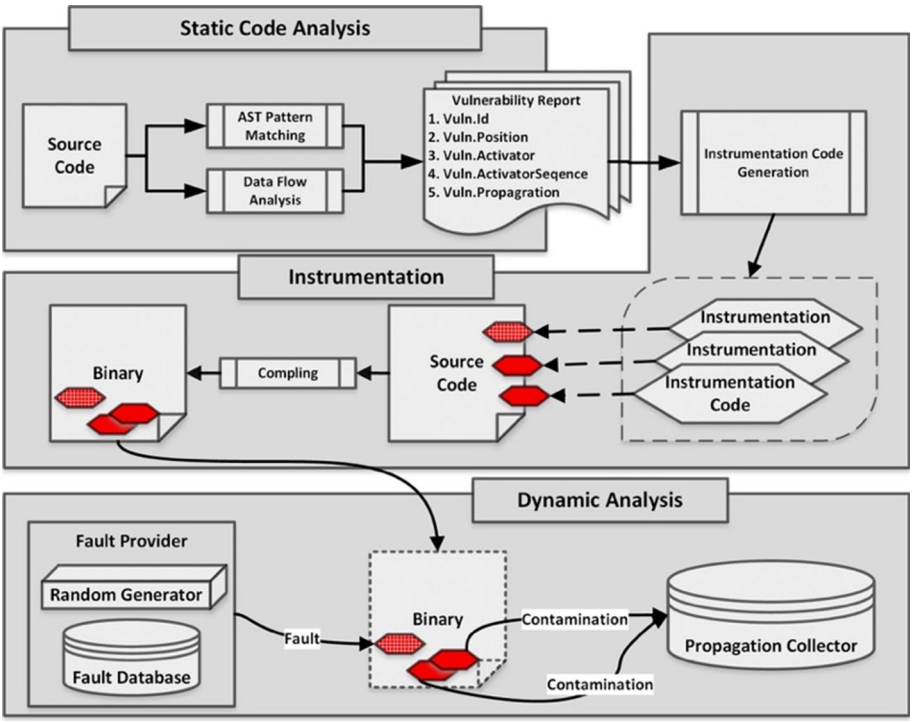

Fig. 2 The proposed method for vulnerability detection

6.1 Phase 1: Static Code Analysis

Static code analysis inspects the source code itself without execution. The proposed method performs two static code analysis, the first is pattern matching using abstract syntax tree, and the other is data flow analysis. The former aims to detect the vulnerability existing in the source code, and the latter aims to check whether the attacker can access the detected vulnerability.

Pattern matching using abstract syntax tree first parses the source code, and expresses the source code in the abstract syntax tree form. This parsing tree is used to detect vulnerability through pattern matching. The vulnerability pattern used here is defined beforehand. If the correspondent vulnerability pattern is found in the target, this analysis gives information of the identity of vulnerability (Vuln.Id) and the position of vulnerability (Vuln.Position). Vuln.Id is the name of the vulnerability pattern correspondent with the target program's source code, and the Vuln.Position is the code line number in which the vulnerability exists in the target program's source code.

And to exploit a vulnerability, the attacker need to be able to access the vulnerability through external interface or such [21]. And to succeed in an attack, the vulnerability used by the attacker should influence the software or the system. For example, it is hypothesized that the web application contains a function with command injection vulnerability. If the value of argument of this function is used as an external input value, the attacker can do command injection attack through the malicious input value. But if the input value is not the value of argument, the attacker cannot input the malicious data, and so the command injection attack is impossible (this is when the attacker cannot access the vulnerability). And even if the value of argument of this function can be manipulated, if affecting is impossible such as not being able to operate malicious command due to privilege, command injection attack is impossible. So the proposed method performed data flow analysis to check the accessibility of vulnerability and to trace the effect of vulnerability. The Fig. 3 shows the accessibility and effect of vulnerability through data flow diagram.

Vulnerability is triggered by activator. And when more than one sequence of activator is linked to the external entity which is the attacker, the attacker can use the vulnerability. And the effect of the activated vulnerability is peppered through the flow of data. Data flow

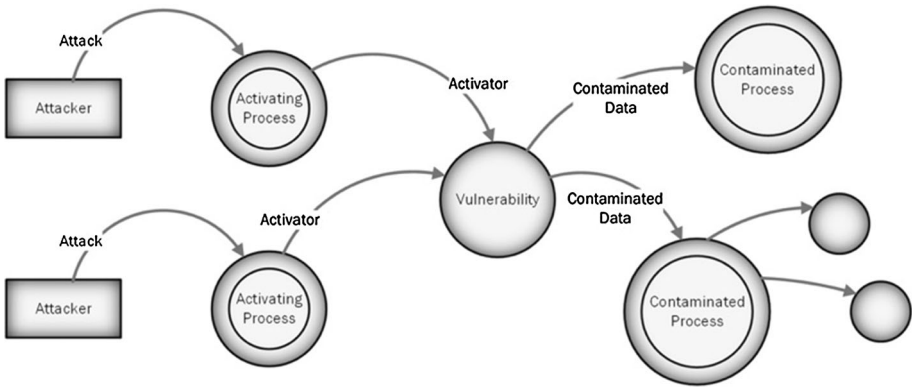

Fig. 3 Data flow diagram focused on vulnerability

analysis can distinguish the activator of vulnerability (Vuln.Activator) and the Sequence of Activators (Vuln.ActivatorSequence). And the Vuln.Propagation can be predicted. The information acquired in phase 1 is in Table 1.

6.2 Phase 2: Instrumentation

Instrumentation stage is the advance preparation stage to verify the vulnerability detected through static code analysis. This preparation stage is for combining the static code analysis with the dynamic analysis. And this stage is operated based on the vulnerability information acquired through static code analysis. Instrumentation is a technique to insert code with particular purpose on the original code. The proposed method uses this technique to inject simulated faults into the target program and to monitor the effects of the injected fault.

The instrumentation code that injects fault is injecting error into the vulnerability. The faults injected into activators of the vulnerability identified from static code analysis. Among the vulnerability activators, the activator included in the activator sequence linked to an external entity(attacker) becomes the fault injection target. This is because for the attacker to exploit the vulnerability, the activator needs to be accessible to the attacker, so that manipulation can be done. The fault data is either supplied by the existing database, or created through random generator if no fault data is defined. The operation of monitoring code is to observe the impact propagation of the injected fault. Therefore, this code logs the state of the contamination, which is a component of the set of the activated vulnerability's propagation gained from static code analysis. Contamination is comprised of contaminated process and contaminated data, but the information that actually gets the logging is the contaminated data. Contaminated data is the input/output data of contaminated process. Table 2 is the operation of instrumentation code simply expressed into pseudo code.

Instrumentation code is various according to object and situation. And this code must be executable when inserted into the original source code of program. Therefore, it must be written in accordance with the syntax and semantics of a programming language. Therefore, when the tester manually writes the code, it takes a lot of time. So in this research, a method of creating instrumentation code automatically based on the defined model beforehand is used. If the instrumentation code is automatically created, the time for writing the code can be saved. And the model defined one time can be reused. And in this research, a template based approach is used to define the model. A template based approach is used wherein a template specifies a text template with placeholders for data to

Table 1 Acquired vulnerability's information in phase 1

Analysis	Acquired information
AST pattern matching	Identity of vulnerability (Vuln.ID)
	Position of vulnerability (Vuln.Position)
Data flow analysis	Activator to trigger vulnerability (Vuln.Activator)
	Sequence of activators to trigger vulnerability (Vuln.ActivatorSeq{Vuln.Activator1, Vuln.Activator2, Vuln.Activator3…})
	Set of the activated vulnerability's propagation (Vuln.Propagation{Contamination1, Contamination2, Contamination3…})

Table 2 Pseudo code of instrumentation code

Instrumentation Code	
Objective	***Pseudo Code***

Fault Injection	**function** faultInjection (vuln){
	IF vuln.ActivatorSeq **IS** accessible **BY** Attacker **THEN**
	IF vuln.Id already exists **IN** database **THEN**
	GET faultData **FROM** database;
	SET vuln.Activator **TO** faultData;
	ELSE
	GET faultData **FROM** randomGenerator;
	SET vuln.Activator **TO** faultData;
	ENDIF
	ELSE
	print “Not exploitable”;
	ENDIF
	}

Propagation Monitor	**function** monitor(vuln) {
	IF faultInjection(vuln) **IS NOT** “Not exploitable” **THEN**
	FOR each contaminations **IN** vuln.Propagation
	Logging contaminatedData;
	Logging contaminatedProcess; //(Optinal)
	ENDFOR
	ENDIF
	}

be extracted from models. So the template specification expresses instrumentation code, and the instrumentation code is automatically created based on this specification. So this specification and transformation needs to be defined. MOF Model to Text Transformation Language (MOFM2T) is a specification of Object Management Group (OMG) for model transformation language. This is especially relatively well defined standard that can be used to model-to-text transformation. Therefore, in this paper MOFM2T is used to create instrumentation code from instrumentation model.

After the instrumentation code generation is over, the instrumentation code required to perform dynamical analysis based on static code analysis is all created. Then the original source code of the target program is backed up, and instrumentation code is inserted. After that target program that can be executed through compiling, linking and such process that is needed to execute a program is formed. Therefore, instrumentation process is comprised of 4 steps, generating instrumentation code, backup the original code, inserting instrumentation code into the original code, and constructing the new executable.

6.3 Phase 3: Dynamic Analysis

The execution of target program is done at the dynamic analysis phase. And the code for analysis is already imbedded in the target program by the former phase (Phase 2). During runtime, the instrumented codes are executed. As mentioned formerly, instrumentation code injects fault that can activate the vulnerability, And the effect of the injected fault is monitored. If the vulnerability that was detected by static code analysis gets activated due to the fault injection, this vulnerability can be used for real attacks. Seeing this from the attacker’s point of view, the activator used to activate the vulnerability becomes the entry

point (attack point). And the fault data is malicious data used for the attack. Also the effect is the result gained from successive attack using the activated vulnerability.

As dynamic analysis through fault injection can have more than one test case, the target program can be executed repeatedly. Therefore, it has a collector to save the analysis materials produced by every iteration. This collector saves the contamination states which the instrumentation code monitors. As the saved data is the impact propagation of injected fault, this collector is called propagation collector. Opposed to this propagation collector, there is also a fault provider which provides fault data. This provides the instrumentation code with fault data. The fault data may be from the fault database, or it can be created from the random generator. So if the fault values from this fault provider are all used, iteration of the execution for analysis is over. And as a result of whole iteration, if one or more fault value activated the vulnerability, this vulnerability can be considered an exploitable vulnerability. And this vulnerability is the true positive of the vulnerabilities detected from the static code analysis.

6.4 Comparison of the Different Hybrid Proposals

The approach proposed in this paper is a composition of static and dynamic analysis techniques. Therefore, in this section, we review the related works (Table 3) that are combined with these two types of analysis. Balzarotti et al. [22] combine static and dynamic analysis techniques to identify faulty sanitization procedures that can be bypassed by an attacker. It identified novel vulnerabilities that stem from incorrect or incomplete sanitization. Halfond et al. [23] proposed a novel technique to counter SQL injection. The technique combines conservative static analysis and runtime monitoring to detect and stop illegal queries before they are executed on the database. Rawat et al. [24] presents a hybrid approach for buffer overflow detection in C code. The approach makes use of static and dynamic analysis of the application under investigation.

Existing researches combining static and dynamic analysis mostly uses dynamic analysis to verify results from static analysis. Therefore, the way proposed which uses dynamic

Table 3 The existing hybrid proposals

Proposals	Binding sequence	Static technique	Dynamic technique	Main purpose
Balzarotti et al. [22]	Pre-static and post-dynamic analysis	Data flow techniques to identify the flows of input values from sources to sensitive sinks	Fault injection to inject strings corresponding to possible XSS and SQL injection attacks to dynamically execute parts of the analyzed applications	To analyze the correctness of the sanitization process
Halfond et al. [23]	Pre-static and post-dynamic analysis	Building a conservative model of the legitimate queries that could be generated by the application	Inspecting the dynamically generated queries for compliance with the statically-built model	To counter SQL-injection
Rawat et al. [24]	Pre-static and post-dynamic analysis	Calculating taint dependency sequences (TDS) between user controlled inputs and vulnerable statements	Executing the program along TDSs to trigger the vulnerability by generating suitable inputs	To detect BoF(Buffer over Flow) vulnerabilities

analysis to solve static analysis accuracy problem is similar to existing researches. However, existing researches mostly handle vulnerabilities due to the input of untrusted data. So, dynamic analysis of these focuses on making input of untrusted data. Therefore, these researches proposed efficient ways to analyze vulnerabilities by the use of untrusted data such as SQL injection or buffer over-flow. However, these proposed methods have difficulty in handling various types of vulnerabilities. But as the way proposed in this paper can materialize various and complex executable test suite through the modeling of Source instrumentation and Instrumentation, many vulnerabilities can be tested. In addition, it has reusability because a model once defined can be used again.

7 Implementation and Experimentation

7.1 Implementation

In this paper, to verify the proposed method, a tool was materialized which allows the application of static code analysis to dynamic analysis using instrumentation. Therefore, this tool automatically creates instrumentation code based on static code analysis information. Then the instrumentation code is inserted into the target program's source code. And the executable of the instrumented target program is formed. After that, the vulnerabilities identified from static code analysis are tested using the formed executable whether it is a false positive or not. So this tool must mainly perform static code analysis, instrumentation, and dynamic analysis. And as the proposed method can be applied in the early software development life cycle, it can also be applied to integrated development environment (IDE). So we developed a tool using one of IDE, platform of Eclipse [25]. Especially, Eclipse can add plug-in so that additional function is added, or original function is expanded. So to develop this tool which needs toolchain to link with various tools, Eclipse platform was used.

The static analysis part of this tool used PMD, one of an open source tool. This is JAVA source code analyzer based on ruleset. This tool deciphers the potential problem on the source code which matches the ruleset defined by pattern matching using AST parsing. This does not largely support ruleset detecting security vulnerability. But it can write custom rules using XPath or Java classes. And this tool can comprehend the structure and meaning of source code because it uses AST parsing. So the static code analysis can be accurately conducted. After the vulnerability is detected, code level information can be acquired. And instrumentation should work based on this. Instrumentation was realized by expanding the function of the tool Acceleo [26]. Acceleo is a pragmatic implementation of the Object Management Group (OMG) MOF Model to Text Language (MTL) standard [27]. This tool helps make code generator. Then, dynamic analysis of verifying the vulnerability using instrumented source code. So, open source dynamic analyzer named JUnit was used. This tool is the most famous unit test framework for java language. The aforementioned tools are supported by Eclipse plug-in. And the tool developed in this paper also was made in Eclipse plug-in. The developed tool consists of Instrumentation Code Outline, Instrumentation Preferences for general instrumentation settings, and Source Code Viewer. The Fig. 4 is a tool developed through Eclipse Plug-in Development Environment (PDE).

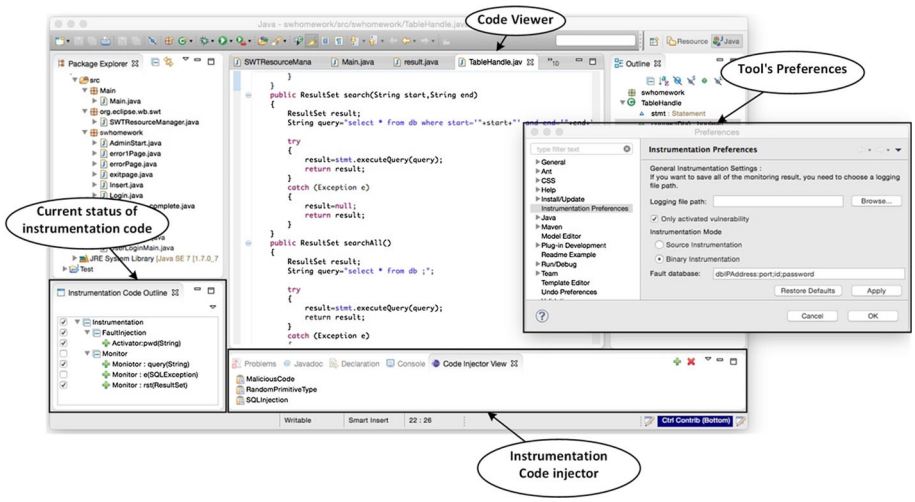

Fig. 4 Screenshot of the developed tool

7.2 Experimentation

We prepared web service which is an example program written in java for experiment. This web service receives ID and password from users for authentication. Also, users can save and inquire their GPS data through this web pages. And this web service has an additional database to save the log of database management system. This web service interacts with three databases in total.

We performed an experiment to detect SQL injection flaw by using tools developed using the proposed method. As the result of the experiment, five SQL injection vulnerabilities in total were detected in the static code analysis phase. And two out of five results which was verified by detecting vulnerabilities through dynamic analysis were vulnerabilities innate in real software. SQL injection vulnerabilities detected by static code analysis occurred one in query disposal side which is used in interaction with user certification database, two in query disposal side used in interaction with GPS database, and two in query disposal side used in interaction with logging database. Of these, the vulnerability that was verified from the dynamic analysis that exists in the real software and that can be used for an attack appeared at the interaction of location database. Rest were false positives. Analyzing these false positives, interaction of logging database records log information inside the system, so it does not use the user's input. So the accessibility to the vulnerability does not exist. This was the same for data flow analysis. In addition, the query handling used in the log-in database interaction used prepared statements (parameterized queries) [28], so as the query and data was separate, it could not be used for real attack. On the other hand, vulnerability detected in the query part used in location information database was activated, and string of the fault data used here was 'or'1'='1. And the vulnerability activator that injected this fault value was user-id, and the sequence of this activator was linked to an external entity. And the activated vulnerability could access or manipulate other user's location information. In conclusion, this vulnerability is accessible to attackers, and the activated vulnerability can affect the system.

Table 4 The reduce of false positives

Experiment #	Rate of false positives (%)		Reduced rate (%)
	Static-only detection	Hybrid detection	
1	25.3 (21/83)	15.7 (13/83)	38.0
2	60.9 (39/64)	43.8 (28/64)	28.2
3	33.3 (9/27)	14.8 (4/27)	55.6

7.3 Contemplation of the Reduce of False Positive

We performed our experiments on a suite of tree web services. These web services contain vulnerabilities intended for the experiments. Our experiment attempts to evaluate the improvement by showing the reduce rate of false positive. Table 4 shows how much the proposed hybrid vulnerability detection is able to reduce false positives.

In Table 4, the reduced rete of false positive is a rate in the percentage showing the reduced amount of false positive rate deciphered by hybrid approach in relation to static-only approach. 40.6 % of false positive rate's reduction in average has been found through dynamic analysis of hybrid approach. In addition, the deviation of false positive rate in the three experiments occurred due to the difference of each system's vulnerability type and distribution. This happens due to the difference of code each target system has.

8 Conclusions and Future Work

The main reason for making many false positives that are made by static vulnerability detection technique is the absence of program operation information. Therefore, the proposed method verifies vulnerability, which is detected by the static code analysis through dynamic analysis. Also, instrumentation technique is used for applying the result of the static code analysis to dynamic analysis. As a result, of the many vulnerabilities acquired through static code analysis, vulnerabilities that can be used in real attack could be verified. This method can check out whether attackers are able to access vulnerabilities though entry point, and trace the effect that influence software and system when vulnerabilities are used for attacking.

The proposed method injects the fault by manipulating the data in data flow sequence of vulnerability, instead of injecting the fault to external inlet. Therefore, it is easy to make error that activates vulnerabilities. The reason is that in order to activate vulnerabilities by manipulating the external input, you have to know target program's execution path but also how to manipulate input data. Without knowing this information, random data should be injected repeatedly. However, if the input data format is complicated or software and system is complicated, the difficulty of generating many test cases is occurred. In this regard, the error injecting method that is used in the proposed method is efficient. And since the developed tool inject errors by using source instrumentation technique, it is possible to inject various errors that could be generated by programing language which is used in target program development. And since this tool generates automatically based on the model that defined by instrumentation code in advance, testers don't have to make the code manually to inject errors. And this method is complementary because it is a combined form of dynamic analysis and static analysis. Imprecision of static analysis is supplemented

by dynamic analysis, and difficulty of generating many test cases of dynamic analysis is supplemented by static analysis. Therefore, vulnerability detection technique that is relatively precise and does not spend much time in analysis is proposed by combining static analysis and dynamic analysis. However, the proposed method is not perfectly improved in the precision side. The proposed method can identify the false positive among the vulnerabilities that is detected by static analysis, but false negative that is not detected by static analysis can not be identified. It is because this method is verifying vulnerabilities detected by static analysis, by dynamic analysis. And since the developed tool use source instrumentation, overhead of recompiling instrumented source code is occurred. In this paper, the method that detects vulnerabilities by using static analysis and dynamic analysis is identified reasonable as a method to reduce false positives through developed tool. Therefore, in the future study the method to identify false negative that is a threshold of this tool will be researched, the way to use binary instrumentation to reduce overhead will be found. In addition, a method to write this model in a diagram to easily write instrumentation model will be found.

Acknowledgments This work was supported by the ICT R&D program of MSIP/IITP. [R0101-15-0144, (EXOBRAIN-4)] development of autonomous intelligent collaboration framework for knowledge bases and smart devises] and "employment contract based master's degree program for information security" supervised by the KISA (KOREA INTERNET SECURITY AGENCY) (H2101-14-1001).

References

1. National Institute of Standards and Technology (NIST). (2014). National vulnerability database. Retrieved September 28, 2014. http://nvd.nist.gov.
2. Dhamankar, R., Dausin, M., Eisenbarth, M., King, J., Kandek, W., Ullrich, J., & Lee, R. (2009). The top cyber security risks. Tipping Point, Qualys, the Internet Storm Center and the SANS Institute faculty, Tech. Rep.
3. Gopalakrishna, R., Spafford, E., & Vitek, J. (2005). Vulnerability likelihood: A probabilistic approach to software assurance. *CERIAS, Purdue Univeristy Tech. Rep, 6*, 2005.
4. Vassilaras, S., & Yovanof, G. S. (2010). Wireless innovations as enablers for complex & dynamic artificial systems. *Wireless Personal Communications, 53*(3), 365–393.
5. Garitano, I., Fayyad, S., & Noll, J. (2015). Multi-metrics approach for security, privacy and dependability in embedded systems. *Wireless Personal Communications, 81*(4), 1359–1376.
6. Gladisch, A., Daher, R., & Tavangarian, D. (2014). Survey on mobility and multihoming in future internet. *Wireless Personal Communications, 74*(1), 45–81.
7. McGraw, G. (2006). *Software security: Building security in* (Vol. 1). Boston: Addison-Wesley Professional.
8. Chess, B., & McGraw, G. (2004). Static analysis for security. *IEEE Security and Privacy, 6*, 76–79.
9. Wheeler, D. (2006). Flawfinder home page. Web page: http://www.dwheeler.com/flawfinder.
10. Viega, J., Bloch, J. T., Kohno, Y., & McGraw, G. (2000). ITS4: A static vulnerability scanner for C and C++ code. In *Computer Security Applications, 2000. ACSAC'00. 16th Annual Conference* (pp. 257–267). IEEE.
11. Copeland, T. (2005). PMD applied. https://pmd.github.io. Accessed 19 Aug 2015.
12. Zhang, J. (2011). A mobile agent-based tool supporting web services testing. *Wireless Personal Communications, 56*(1), 147–172.
13. Hsueh, M. C., Tsai, T. K., & Iyer, R. K. (1997). Fault injection techniques and tools. *Computer, 30*(4), 75–82.
14. Source code instrumentation overview at IBM website, http://www-01.ibm.com/support/knowledgecenter/#!/SSSHUF_8.0.0/com.ibm.rational.testrt.doc/topics/cinstruovw.html.
15. Huang, J. C. (1978). Program instrumentation and software testing. *Computer, 4*, 25–32.
16. Introduction to instrumentation and tracing at Microsoft developer network website, https://msdn.microsoft.com/en-us/library/aa983649(VS.71).aspx.

17. Luk, C. K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., & Hazelwood, K. (2005). Pin: Building customized program analysis tools with dynamic instrumentation. In *ACM Sigplan Notices* (Vol. 40, No. 6, pp. 190–200). ACM.
18. Bala, V., Duesterwald, E., & Banerjia, S. (2000). Dynamo: A transparent dynamic optimization system. In *ACM SIGPLAN Notices* (Vol. 35, No. 5, pp. 1–12). ACM.
19. Mens, T., & Van Gorp, P. (2006). A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science, 152*, 125–142.
20. Object Management Group. http://www.omg.org.
21. Mell, P., Scarfone, K., & Romanosky, S. (2006). Common vulnerability scoring system. *Security & Privacy, IEEE, 4*(6), 85–89.
22. Balzarotti, D., Cova, M., Felmetsger, V., Jovanovic, N., Kirda, E., Kruegel, C., & Vigna, G. (2008). Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on* (pp. 387–401). IEEE.
23. Halfond, W. G. J., Choudhary, S. R., & Orso, A. (2011). Improving penetration testing through static and dynamic analysis. *Software Testing, Verification and Reliability, 21*(3), 195–214.
24. Rawat, S., Ceara, D., Mounier, L., & Potet, M. L. (2013). Combining static and dynamic analysis for vulnerability detection. arXiv preprint arXiv:1305.3883.
25. Eclipse. https://www.eclipse.org/.
26. Acceleo, Eclipse plugin. http://www.eclipse.org/acceleo/.
27. MOFM2T. http://www.omg.org/spec/MOFM2T/1.0/.
28. Thomas, S., & Williams, L. (2007). Using automated fix generation to secure SQL statements. In *Proceedings of the Third International Workshop on Software Engineering for Secure Systems* (p. 9). IEEE Computer Society.

Seokmo Kim received the B.S. degree in Computer and Information Communications Engineering from Hongik University, Korea in 2014. He is currently a M.S. candidate in Dankook University. His research interests are in the areas of Security Analysis, Testing and Secure SDLC.

R. Young Chul Kim received the B.S. degree in Computer Science from Hongik University, Korea in 1985, and the Ph.D. degree in Software Engineering from the department of Computer Science, Illinois Institute of Technology (IIT), USA in 2000. He is currently a professor in Hongik University. His research interests are in the areas of Test Maturity Model, Embedded Software Development Methodology, Model Based Testing, Metamodel, Business Process Model and User Behavior Analysis Methodology.

Young B. Park received the M.S. and Ph.D. degree from the department of Computer Science, N. Y. Polytechnic University (NYU-Poly) in 1991. He is currently a professor in Dankook University. His research interests are in the areas of Intelligent Software Engineering, Automatic Software Testing, Software Development Process Enhancement and Software Refactoring.