Using Model Checking to Detect SQL Injection Vulnerability in Java Code

ISSN (e) 2520-7393
ISSN (p) 2521-5027
Received on 15th Dec, 2019
Revised on 28th Dec, 2019
www.estirj.com

Muhammad Adil[1], Irshad Ahmed Sumra[2]

Department of Computer Science, Bahria University, Lahore, Pakistan.
Email adil.concordia@gmail.com[1] isomro28@gmail.com[2]

Abstract: Several techniques have been proposed for the detection of SQL injection vulnerability detection in the source code. These techniques include the analysis of the code to detect SQL injection vulnerabilities. The code of the software system under consideration can be tested for SQL injection vulnerabilities and the tester can be notified. The existing techniques either lack the ability to detect SQL injection vulnerability or provides greater number of false positives (detection of non-existent SQL injection vulnerabilities).In our research proposed SQL injection vulnerability detection in Java code through model checking. The code is transformed into intermediate representa- tions (graphs), which then are converted into model. The model is checked against the specifications that are created according to the requirement (in our case SQL injection vulnerability detection). If the model satisfies the specifications, the code represented by the model is concluded to be safe from SQL injection vulnerability. We have used constant propagation analysis (CPA) for the purpose of expression evaluation for constant variables used in the test programs. We have also proposed algorithms to create necessary graphs from Java byte code to act as an intermediate representation of the code. Results have shown that the number of false positives are reduced while increasing the number of true negatives for the OWASP Benchmarks. The number of true positives for our implemented approach is consistent for the bench- mark test cases that are handled by the implementation of our approach.

Keywords: *SQL injection vulnerability, Java, Constant propagation analysis, OWASP Benchmarks.*

1. Introduction

The importance of software systems has been growing for the past few decades, a large number of software he systems are developed and their usefulness cannot be denied. Enterprise resource planning, business intelligence and internet based transaction system are only few examples from a long list of software applications (Gorton, Brown, and Banks, 2004) [1]. The need of fulfilling the changing dynamics towards the software industry leads to the growth in software development, this growth brings software management into the core of corporate strategies (Probert et al., 2005) [2]. Security defects in software can be damaging in several aspects including financial loss, security risks etc. this is caused by downtime, disruption and confidentiality breaches in the system, which can be caused by security defects. Security defects can be introduced in a software system due to human errors or bad coding practices (etc.).

The software that contains security defects is said to be vulnerable attacks, and such security defects are often referred to as software security vulnerability (Telang and Wattal, 2007) [3]. The detection of software vulnerabilities in a software system is crucial, these vulnerabilities can be performed either manually or automatically. The goal of both (manual and automatic vulnerability detection) is to identify vulnerabilities in a software system that is the identification of vulnerabilities that in the later process can be dealt with. However, automated software vulnerability detection is more frequent choice as compared to manual vulnerability detection, the reason being less execution time and lower costs (Antunes and Vieira, 2009) [4]. In the following section, we discuss automated vulnerability detection. Automated Vulnerability Detection to avoid detection, the developers must apply best coding practices.

The security review of a software system is also critical i.e. applying penetration testing and use of code analysis tools. However, in most cases the user requirements are focused more than the security risk analysis and numerous developers are not specialized in software security. Furthermore, the time constraint of the development life cycle limits the in depth test of security the identification and elimination of the security vulnerability can be time-consuming. By auto-mating the process of security vulnerability detection, one can improve the time aspect of the security vulnerability test.

To automate the vulnerability detection, different tools are used, these tools have different capabilities regarding the vulnerability detection. Code analysis during software development and evolution, the most reliable and precise description of the software is its source code. With the source code the high-level architectural design of a software system may also be useful for the perception of the software system. For example a high-level architectural design may be useful for localization of the components which should go under modification or testing, it can also provide the magnitude of the change and how it can affect the whole system. With all the advantages of the architectural design, a diagram may be inconsistent with the code, leaving the code as the best description of the software system (Vanciu and Abi-Antoun, 2013) [5]. The best way to counter this drawback is to extract the architectural design from the code itself. Different diagrams (e.g. class diagram (Tonella, 2005) [6], control flow graph ((Lanubile and Visaggio, 1997) and data flow graph (Lanubile and Visaggio, 1997)) [7] can be extracted from the code. These diagrams can be useful in several ways and are consistent with the code (Vanciu and Abi-Antoun, 2009) [8].

The code analysis can also be used for other purposes including vulnerability detection. This can be done by either manually analyzing the code or by making use of an automated analysis tool, the later is less time consuming and provides an effective way of software vulnerability detection. However, the effectiveness of automated analysis tool may depend on the tool in use. Different tools have better performance in different types of vulnerability. Hence, there is a need to use some kind of standards to measure the effectiveness, efficiency and correctness of the vulnerability detection of the code analysis tools.

The code analysis of a software system may be categorized into two broad categories i.e. static and dynamic code analysis. Static code analysis is performed in non-runtime environment in contrast with the dynamic code analysis. A number of tools are available for the automation of code analysis. These tools may perform either static or dynamic code analysis, or may perform both kinds of code analysis. The static code analysis, analyzes the code statically and tests the code for the possible scenarios that may come into play during the run time.

The dynamic code analysis tools on the other hand executes the software system for different scenarios. Static analysis, with its white box visibility is more thorough than the dynamic approach and in some cases may prove to be cost effective as it can be applied in early phases of software security vulnerability testing. In contrast, the dynamic code analysis tests the parts of program that are under execution during the test. But the dynamic analysis approach has its own advantages over the static analysis as it can be used to determine the efficiency, memory consumption and other aspects of the software system under observation, which are not possible to determine through static analysis.

CERT Standards and SQL Injection Vulnerability as the different defects or vulnerabilities in a software system may be of different categories, priority, likely- hood, remediation cost. The standardization of the vulnerabilities will provide an easy and effective way to prioritize the detection of vulnerabilities. The SEI CERT Coding Standard is a software coding standard for programming languages, developed by the CERT Coordination Center to improve the safety, reliability, and security of software systems (Abi-Antoun and Aldrich, 2009).

CERT standards provides coding and vulnerability standards for different programming languages. CERT provides coding and vulnerability standards for most common languages like C, C++ and Java. These standards allow a programmer to follow the standard set of rules during the software system development, moreover it also categorizes the security vulnerabilities according to their priority, likelihood, remediation cost etc.

SQL Injection may cause a serious threat to Web and Desktop applications. It allows the attacker to obtain unrestricted access to the database. The SQL Injection threat is more damaging in cases where the database has confidential information like Credit Card information, or in security critical software system like defense systems. SQL injection vulnerabilities have been described as one of the most serious threats for Web applications.

The SQL Injection threat if executed may cause identity theft, loss of confidential information, and fraud. The SQL Injection attack (SQLIA) can be categorized according to the intention of the attacker. The intention may vary from data extraction, data modification, bypassing authentication and much more. The SQL Injection Vulnerability (SQLIV) in some cases might be the cause of SQL Injection at- tack. Hence, it is of significant importance to counter the SQL Injection Vulnerability in a system under development or testing before its release.

This paper is organized as pursues: Section 2, talks about some related works. In segment 3, examines the various calculations and information accumulate assets. Area 4, introduces the outcome and discourse in detail. At last, area 5, shows the end and future research.

2. Related Work

In this section we describe the related work and approaches that are related to taint analysis. We have mainly focused on taint analysis of the program for the purpose of security vulnerability detection. We consider related work that is associated to identify security vulnerability in different languages, as the mentioned methods for one specific language can be applied to any other language.

The main focus is the identification of taint source, taint kill and taint sink and the algorithms and techniques that are used for taint analysis. Tripp et al, 2009 proposed an approach for taint analysis of Java applications, by construction of call graph and performing pointer analysis. The pointer analysis adds the call string methods to the factory (local storage to call string methods). The pointer flow precision is taken into consideration. The methods calls for the ones that are contextually sensitive are analyzed. The taint source and taint sink are analyzed with one level of method calls. This approach makes use of hybrid thin slicing (Gallagher and Binkley, 2008), the objects are abstracted and an abstracted model is created which is to be analyzed instead of real Java program those statements that are data dependent are considered and taint carriers are identified and tracked in the model.

A set of objects that are reachable by an object are constructed and those objects are used for the identification of taint flow, this taint flow is used for vulnerability detection [9, 10]. Livshits and Lam et al, 2005 presented a method for taint object propagation to check security vulnera- bilities by identifying taint object and sink objects, and performing static taint analysis of a program. If the sink object is tainted, then the program contains vulnerability. The string values, when they are assigned to each other, the assignment of the tainted value is in effect too. The string methods that use tainted values are matched with the ones that use a tainted variable value either partially or completely. These string methods if used transfer the taint to the variable defined, also direct assignment of tainted variable to another variable also passes the taint. And finally the sink object (the object that is sinked) is checked for taint.

If the sink object is tainted then this approach shows that there exists a security vulnerability within the input program of static taint analysis. Otherwise, it is concluded security vulnerability does not exist [11]. Yan, Ma, and Wang et al, 2017 presented a backwards approach for detecting web application vulnera- abilities in PHP code. The proposed approach uses static code analysis for taint analysis and vulnerability detection. It also considers the sanitation of the data (i.e. Removal of taint or taint kill) and considers that the variables which are used in sanitation functions, have their taints removed during the taint analysis pro- cess. The execution paths of the codes are also identified for that code and are considered to be used in the process.

Taint sources and taint sinks for different vulnerability types are identified and then vulnerabilities are detected while taking into consideration the sanitization of the data through sanitization functions such as HTML entities() available in PHP [12]. Ming et al, 2016 developed a taint analysis tool that uses hybrid taint analysis. This tool keeps log of information that is extracted from the source code, which decouples the taint analysis from the source code. Once data is extracted from the code the taint analysis can be performed offline (the source code is not required again for analysis). Once information is extracted from the code, the tool uses multi-threading for security vulnerabilities detection while using queues for processes, while taking into consideration the dependencies in the data flow. The tool uses hybrid approach, which allows the use of dynamic and static taint analysis and put the advantages of both for the identification of security vulnerabilities in a program [13].

Cao et al., 2017 also proposed an approach for static code analysis in which the program is trans- formed into an intermediate representation which is to be used for taint analysis. This approach makes use of Control Flow Graph (CFG) and Abstract Syntax Tree (AST) as the intermediate presentation of the code. These intermediate representations are created by performing lexical and grammatical analysis. It takes output of the PHP parser in the form of AST that is then transformed into a CFG. As the identification of the taint source, all external input data for a program are considered tainted, these sources include but are not limited to the parameter values from the interface, cookies or session information. Then a flow sensitive forward taint analysis is performed on the CFG for the identification of sinking of tainted variables.

If a tainted variable is sinked, then it is concluded that there exists a vulnerability in the source code. This technique is successfully used for the identification of Reflected-XSS, Stored-XSS and SQL bland injection but is unable to identify SQL injection vulnerability. The reason being it does not consider branching in the graph that are caused by conditional statements [14]. Chen, Wang, and Zhang et al, 2011 also make use of Control Flow Graph (CFG) for taint analysis, the approach proposed a solution to branching problem in the code. Where CFG is created and different branches act as outgoing edges of a block, this occurs due to the use of branch statements (i.e. if-else, switch statements).

This approach hacks on run-time that if the tainted data is used in illegitimate way then a signal for an attack is triggered. During this dynamic taint analysis only those paths are executed which are considered to be true during the execution. Hence avoiding all the other paths that might or might not have triggered the signal of an attack. And that is also the main disadvantage while using taint dynamic taint analysis. As this approach does not identify other paths, which might have vulnerability and might allow the user to take advantage of and help in execution of an attack [15].

Li, Zhang, and Wang et al, 2014 describes an approach for online dynamic taint analysis in binaries to encounter security vulnerability. A point-to set is created, which stores the information about the pointer and memory location that the pointer refers. This approach suggests that the memory location which is referred by a pointer is tainted rather than the actual variable. Hence, the point-to taint analysis is performed for the propagation of taint among memory taint values, the precision of this taint analysis is on byte level in memory, which is precise enough for point-to taint analysis but is not effective in terms of memory consumption and is not efficient while processing [16]. Fehnker, Huuck, and Rödiger, 2011 describes a model checking based algorithm to encounter security threats in embedding system. Data flow and taint analysis is performed by creating a model from control flow graph (CFG), which is the abstraction of the actual program. The CFG created by this approach also contains the definition and usage of a variable. The taint model is then created from the CFG and CTL (computational temporal logic, a logic to describe system properties or behavior) is used to verify the model. If the model satisfies the CTL specification then it is concluded by this approach that there is no vulnerability. The shortcoming of the approach presented in this paper is that it handles only one variable. Furthermore the proposed approach does not handle taint transfer from one variable to another [17].

3. Methodology

Detectors in FindSecBugs are the classes that are used to detect a specific vulnerability. FindBugs.xml is the file where all the detectors and Engines are listed to be used by FindBugs. We have introduced our own detector class called BetterSQLInjectionDetector for the purpose of implementation of our approach. As FindSecBugs deals with Byte code. The algorithm we have proposed is for Java byte code. We have proposed an Algorithm to create taint graph with the DFG information while traversing CFG.

Algorithm 1 is used to create a Data Flow Graph (DFG) from a Control Flow Graph (CFG). Although Algorithm 1 this algorithm is generic.

Algorithm to Create DFG from CFG

Input CFG cfg

Output DFG dfg

```
1:  procedure CREATEDFG
2: df g ←EMPTY
3: currN ode ← root of cfg
4: add currN odetonodeProcessList
5:    while nodeProcessList IS NOT Empty do
6:    currN ode ← Pop nodeProcessList
7:    newDFttN ode ←Empty

8:    defs ← getDefinitions from currNode
9:    uses ← getUses from currNode
10:   for each def ∈defs do
11:   add deftonewDFGNode
12:   for each use ∈uses do
13:   add usetonewDFGNode
14:   add newDFttN odetodfg
```

Proposed the create Kripke structure while traversing the taint graph. While traversing the taint graph we have proposed Algorithm 2 to create Kripke structure. The edges in the taint graph are translated to transitions in the Kripke structure. The nodes of a graph are stored as state of Kripke structure.

Algorithm to Create Kripke Structure

Input Taint Graph taint Graph, List VarList

Output Kripke kripke

```
1:  procedure CREATEKRIPKE (tain graph)
2:    inputSymbol ←nn-D nn stands for next   node
3:    if ¬ inputSymbol already Added then
4:    kripke.addInputSymbol( inputSymbol )
5:    parentN ode ←taintGraph.getRoot( )
6:    while parentNode has Children do
7:    childN ode ←parentNode.nextChild( )
8:    kripke.addState(childNode)
9:    kripke.addTransition(parentNode,        inputSymbol,
childNode)return kripke
```

For the implementation purpose of proposed techniques we have selected FindSecBugs, a static analysis tool for vulnerability detection, which detects vulnerabilities including SQL injection vulnerability (the main focus of our research) in Java byte code. FindSecBugs is the extension of FindBugs and uses the data structures provided by FindBugs, which are ready to be used.

The existing architecture of FindSecBugs consists of three major components that FindBugs uses for the identification of security vulnerabilities. FindSecBugs differs from FindBugs in use that it focuses solely on security vulnerabities, rather than common error and mistakes in Java byte code during analysis. Reason for selecting FindSecBugs is the availability of its complete source code (which is not case in other free tools like SonarQube, whose source code is partially available). Furthermore FindSecBugs has greater overall OWASP score as compared to most common open-source tools.

3.1 Taint Engine

Engine in FindBugs is a part of code that is run after the data-structures are created. TaintEngine in FindSecBugs (Extension of Findbugs) runs with other engines of FindBugs when the extension is added to the FindBugs or Spot Bugs (successor of FindBugs). This taint engine is used by the Detectors that target a specific vulnerability in the code. Without creating our taint engine, we have modified the code in the SQL Injection Detector of the classes.

3.2 Detectors

Detectors in FindSecBugs are the classes that are used to detect a specific vulnerability (like SQL injection vulnerability). The functionality of detectors for each vulnerability is defined in a separate detector class for each vulnerability.

For example SQLInjectionDetector is the detector class for the SQL injection vulnerability. Each detector has a long hierarchy of parent classes. The main parent class from which all other detector classes are derived is Detector interface class.

3.3 Bugs List

Bugs List contains the list of the bugs that need are identified by FindSecBugs. The Bugs in FindSecBugs refers to security vulnerabilities, in contrast to FindBugs where Bugs are considered to be bad practices in the code. The reason for this is that FindSecBugs focuses on security vulnerabilities.

3.4 FindBugs.xml

FindBugs.xml is the file where all the engines, detectors and bugs are listed to be used by FindBugs.

It is important to note that there does not exist any piece of FindSecBugs code where a detector class instantiated.

As these classes are only used by FindBugs / Spot Bugs.If a detector class does contain the definition in the code but is not mentioned in the FindBugs.xml file then the class will not be considered by the FindBugs / Spot Bugs program to be used as detector during the process of byte code analysis by the tool.

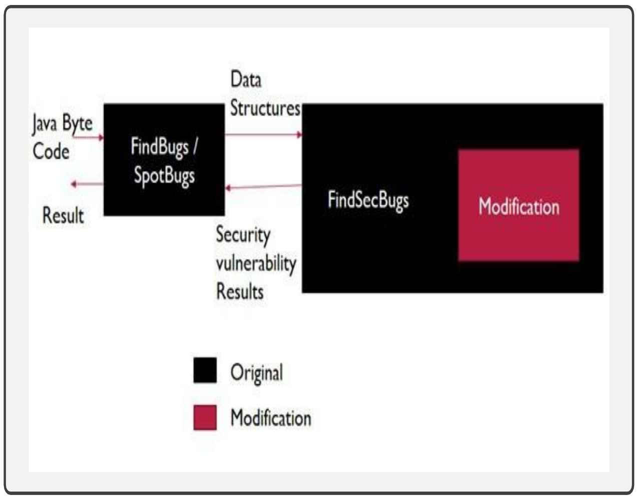

Figure 3.1: Block Figure of FindSecBugs

3.2 Modification of FindSecBugs for proposed approach implementation

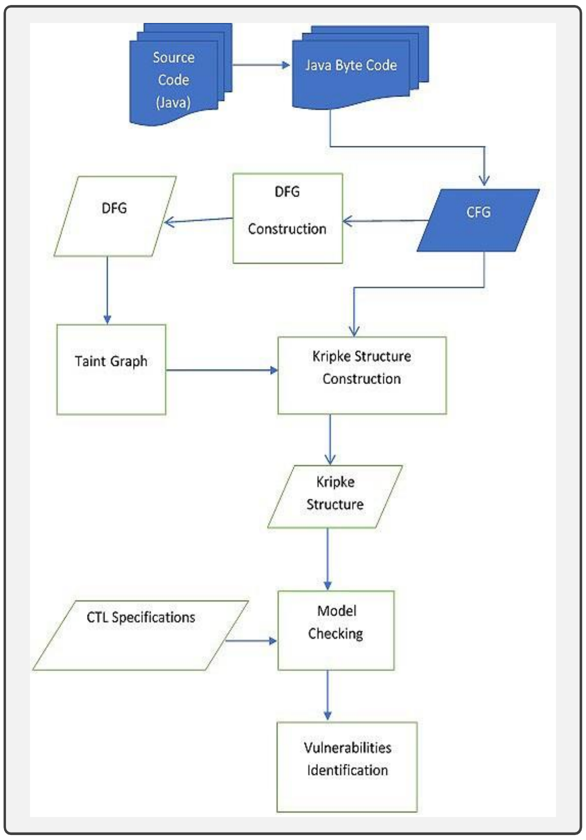

Figure 3.2: Block Figure of FindSecBugs

Our main objective is to apply our proposed approach to the SQL injection vulnerability. As mentioned in the earlier section the class for SQL injection vulnerability is the SQLInjectionDetector class. The class in itself contains only two methods that adds the injection points specified in a text file. The rest of the methods are in its parent's class and are not overridden.

4. Results and Discussion

We have compared the results with different measures. We have considered different variations of the test cases for example skipping the low priority SQLIV. FindSecBugs classifies the SQLIV to be of high priority and low priority depending on the taint sink used. We have considered two measures Youden's Index (that is used by OWASP benchmarks) and Accuracy.

In OWASP Benchmarks total high priority test cases are 2740, out of these test cases the number of test cases with SQLIV is 504, and the number of test cases without for SQLIV test is 2,236 as shown in figure 4.1.

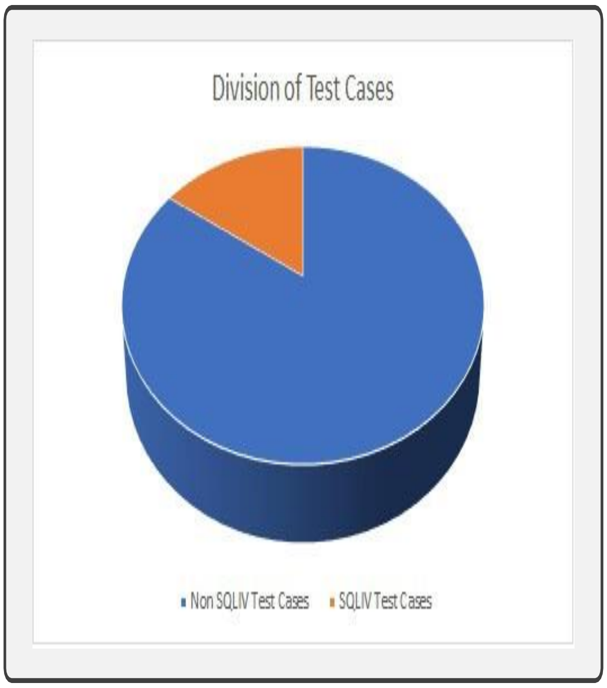

Figure 4.1: Division of Test Cases for SQLIV

Since we have used model checker for determination of SQLIV, it's important to describe some benefits of using model checking for vulnerability detection. The benefit of using a model checker is that the same model can be used to achieve different purposes. For example a model created by our approach uses the constant propagation analysis (i.e. Using variable values instead of variables to evaluate an expression). To take advantage of model checking technique, one can remove the effects of constant propagation analysis by changing only CTL properties of the model checking. Though this technique is very useful in some cases (cases in which we need to use same model to get results with and without constant propagation analysis) our goal for implementation of approach is to lessen the number of false positives, hence we need constant propagation analysis only.

The reason being that the existing system is already removing the effects of constant propagation analysis.

Moreover, similar models can be checked against CTL property of model checking to achieve different goals e.g. SQLIV detection checking and taint return by a code method (both of these functionalities can be achieved by similar models with same CTL specifications).

While considering the test cases for which the model is well formed (i.e. the model is true representation of the code for checking SQLIV) the results are shown in Table 4.1 and is represented graphically in Figure 4.2 and 4.3. The reason for Youden's index to be zero for proposed approach is that specificity to be zero (as false positive rate is one, subtracting it from 1 will return zero). False positive rate is the ratio of false positive to the total candidates that can be false positive.

Table 4.1: Result for OWASP Test Cases for SQLIV (FindSecBugs vs Proposed Approach)

	True Positive	True Negative	False Positive	False Negative
FindSecBugs	205	0	73	0
Proposed	205	73	0	0
FindSecBugs	Sensitivity =1	Specificity = 0	FPR=1	FNR=0
Proposed	Sensitivity =1	Specificity = 1	FPR=0	FNR=0
FindSecBugs	Youdens Index = 0%		Accurac y =	73.74
Proposed	Youdens Index = 100%		Accuracy =	100

In our approach we have created the model of a program, the model is the representation of the given program. The model is created by using the intermediate representation of the code (i.e. Control Flow Graph and Data Flow Graph).

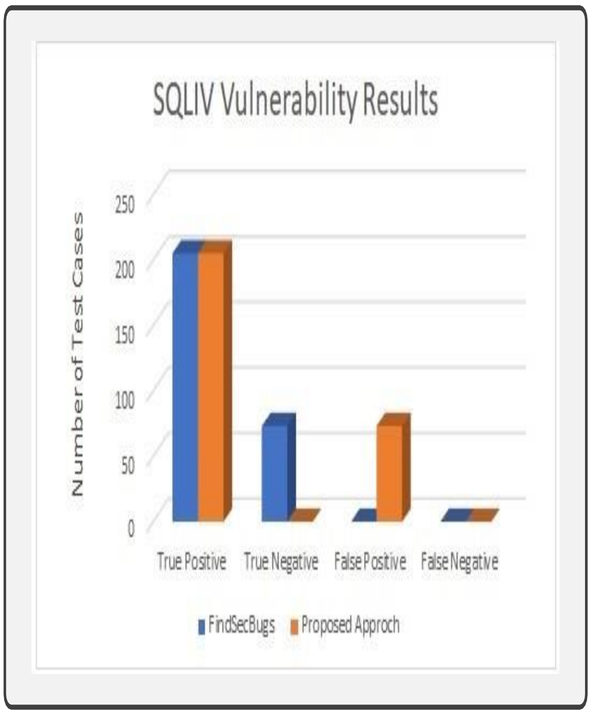

Figure 4.2: SQL Vulnerability Results for Selected Test Cases

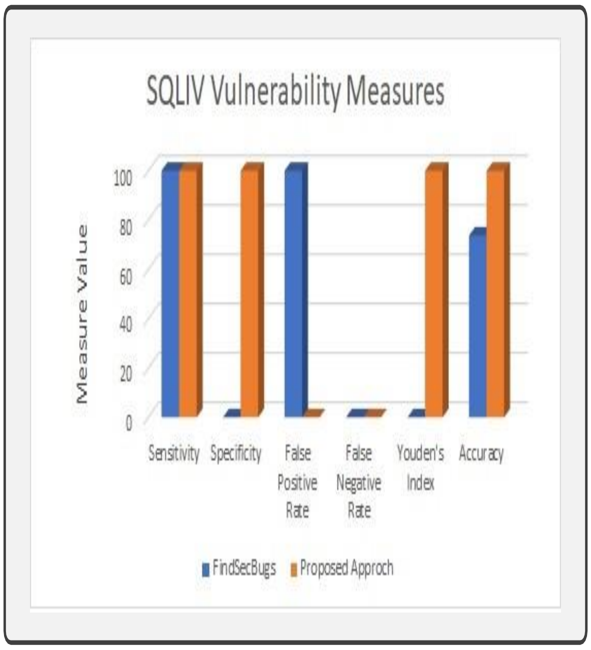

Figure 4.3: SQL Vulnerability Results for Selected Test Cases

The model is then checked against the CTL formulas. We have used following CTL formula.

(¬EF (taint sink) |¬EF (taint) |¬Ett((taint)− > EX((EF ((taint sink))− > (EX(EF (taint kill)))))))

By running our experiments we have concluded that the CTL formula given above works fine for SQL injection vulnerability detection through model checking, the only requirement for the formula to catch SQL injection vulnerability is to that the model must be true representation of the code of the given code, in case the SQL injection is passed without being detected is the limitations in the process of model creation. These limitations in the model creation are normally due to insufficient number of variations that are handled during the model creation. For example handling some variations of String (e.g. String concatenation) in a Java program but not handling other variations like substring ( ) method of the String class. Which will result in creation of a model that does not fulfill our requirements of SQL injection vulnerability detection. We have used the same CTL for different model to achieve different goal. For example if the model represents a SQLIV with taint, taint_kill and taint sink we have used the CTL to identify the SQL injection vulnerability on the created model.

Whereas if a model represents a function of a class in Java program and the requirement is to determine whether or not that function is returning tainted variable (depending on the tainted variables passed to that function during function call), the taint sink action if returned with taint return in the model and CTL specification fulfills our requirements of testing taint return by a function. We have used existing model checker NuSMV as it provides an easy way to analyze a binary encoded model. In this particular implementation of our approach the binary encoded model is sufficient as there are at-most 4 variations that needs to be stored for each node (or state) of the model, which can be represented in 2 bits. Another reason being that Boolean model is faster as compared with the integer model. Although the specification becomes a little more complex (as can be seen by the given CTL formula above).

For the test case selection, used three variations of the test cases from the given OWASP benchmarks. Although OWASP benchmarks does not give priority to these test cases, the existing Find- SecBugs classifies the test cases to be of High Priority vulnerabilities and Low Priority vulnerability, this classification is done based on the taint sink (methods or part of code that access databases) used.

Firstly, we have run both of the tools (FindSecBugs and Proposed Approach) on all of the test cases. Next have run our tests on only high priority SQLIV test cases. And lastly we have run mixed approach (partially proposed approach and partially FindSecBugs, i.e. Proposed approach is run on high priority SQLIV and FindSecBugs is run on low priority SQLIV), which has resulted in highest Youden's Index (the measure used by OWASP for static analysis tool), as the sensitivity and specificity is higher for the mixed approach as compared to the proposed approach. False positive rate that favors lesser value has a greater value for mixed approach (the reason being increase in the false positive), but the False Negative Rate has decreased for the mixed approach as compared to the proposed approach.

5. Conclusion and Future work

Proposed an approach for the taint analysis for SQL injection vulnerabilities. In our approach we have used some intermediate representations of the code CFG, DFG and Taint Graph. These intermediate representations help in the creation of model that is used for vulnerability detection. The specification, which is created to fulfill the requirement of SQL injection vulnerability detection is then checked against the created model to determine the existence of vulnerability in the code (represented by the model). The main advantage of our approach is that our approach allows the testing of all (or selected) paths in the graph through model checking depending on the specification that is checked against the model, while considering the result of evaluation of expression through CPA (Constant Propagation Analysis).Though with any other approach there are some limitations and improvements that can be handled later on. Following sections describe the limitations of our approach.

References

[1] Gorton, M, S Brown, and S Banks (2004). Analysis of the multi-channel software digital content and related service converged business space - a report to the Department of Trade and Industry.

[2] Probert, D et al. (2005). “Developing software content for manufactured products: inside or outside the firm?” In: Proceedings of the R & D Management conference 2005, Pisa, Italy.

[3] Telang, Rahul and Sunil Wattal (2007). “An empirical analysis of the impact of software vulnerability an- nouncements on firm stock price”. In: IEEE Transactions on Software Engineering 8, pp. 544–557.

[4] Antunes, Nuno and Marco Vieira (2009). “Comparing the effectiveness of penetration testing and static code analysis on the detection of sql injection vulnerabilities in web services”. In: Dependable Computing, 2009.

[5] Vanciu, Radu and Marwan Abi-Antoun (2013). “Finding architectural flaws using constraints”. In: Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on. IEEE, pp. 334–344.

[6] Tonella, Paolo (2005). “Reverse engineering of object oriented code”. In: Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on. IEEE, pp. 724–725.

[7] Lanubile, Filippo and Giuseppe Visaggio (1997). “Extracting reusable functions by flow graph based program slicing”. In: IEEE Transactions on Software Engineering 23.4, pp. 246–259.

[8] Abi-Antoun, Marwan and Jonathan Aldrich (2009). “Static extraction and conformance analysis of hierar- chical runtime architectural structure using annotations”. In: ACM SIGPLAN Notices. Vol. 44. 10. ACM, pp. 321–340.

[9] Tripp, Omer et al. (2009). “TAJ: effective taint analysis of web applications”. In: ACM Sigplan Notices.

[10] Gallagher, Keith and David Binkley (2008). “Program slicing”. In: 2008 IEEE International Conference on Software Maintenance. IEEE, pp. 58–67.

[11] Livshits, V Benjamin and Monica S Lam (2005). “Finding Security Vulnerabilities in Java Applications with Static Analysis.” In: USENIX Security Symposium. Vol. 14, pp. 18–18.

[12] Yan, Xuexiong, Hengtai Ma, and Qingxian Wang (2017). “A static backward taint data analysis method for detecting web application vulnerabilities”. In: Communication Software and Networks (ICCSN), 2017 IEEE 9th International Conference on. IEEE, pp. 1138–1141.

[13] Ming, Jiang et al. (2016). “StraightTaint: Decoupled offline symbolic taint analysis”. In: Automated SoftwareEngineering (ASE), 2016 31st IEEE/ACM International Conference on. IEEE, pp. 308–319.

[14] Cao, Kai et al. (2017). “PHP vulnerability detection based on taint analysis”. In: Reliability, Infocom Tech- nologies and Optimization (Trends and Future Directions)(ICRITO), 2017 6th International Conference on. IEEE, pp. 436–439.

[15] Chen, Zhe, Xiao Juan Wang, and Xin Xin Zhang (2011). “Dynamic Taint Analysis with Control Flow Graph for Vulnerability Analysis”. In: Instrumentation, Measurement, Computer, Communication and Control, 2011 First International Conference on. IEEE, pp. 228–231.

[16] Li, Gen, Ying Zhang, Shuang Xi Wang, et al. (2014). “Online Taint Propagation Analysis with Precise Pointer- to Analysis for Detecting Bugs in Binaries”. In: High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC,

CSS, ICESS), 2014 IEEE Intl Conf on. IEEE, pp. 778–784.

[17] Fehnker, Ansgar, Ralf Huuck, and Wolf Rödiger (2011). “Model checking dataflow for malicious input”. In:Proceedings of the Workshop on Embedded Systems Security. ACM, p. 4.