MIS Review Vol. 19, No. 2, March (2014), pp. 39-56
DOI: 10.6131/MISR.2014.1902.03

Detection and Analysis of Security Vulnerabilities in Java

Ch. Aswani Kumar, M. Sai Charitha
School of Information Technology & Engineering, VIT University, India

ABSTRACT: *There are several tools that use techniques like static analysis, lexical analysis etc to detect the vulnerabilities in Java based programs. However still there are vulnerabilities which are not traceable by the available tools. The objective of this paper is twofold. We develop a method to detect vulnerabilities in Java programs. Further we analyze the dependencies among the vulnerabilities using mathematical lattice theory based formal concept analysis. Our experimental result show that the proposed model is able to detect the untraceable vulnerabilities and the dependency analysis is in good agreement with the literature.*

KEYWORDS: *Formal Concept Analysis, Java Programs, Static Analysis, Vulnerability Detection.*

1. Introduction

Vulnerabilities within the program language constructs can inject the vulnerabilities in the applications as well. Poor coding practices also introduce risks into the programs. Generally programs are verified and tested to ensure that they meet the requirement specifications with regard to functionality, quality and performance. However verification of these programs for potential security vulnerabilities is also an equal and major concern. If left undetected, these vulnerabilities would be exploited by the adversaries. Also it is essential to analyze these vulnerabilities to better understand the threats and risks. Dependencies among the vulnerabilities provide insight into occurrence of security attacks. Vulnerabilities are broadly categorized as either implementation bugs or design flaws (Austin, Holmgreen & Williams, 2013).

The primary advantage of Java language lies in its ability in developing and deploying applications in heterogeneous platforms. Java includes several security features like cryptography, authentication, PKI, etc. Despite of these features, Java has several vulnerabilities that enable adversaries to cause potential attacks. Since several vulnerabilities are specific to Java language, traditional detection and mitigation methods does not work (Garber, 2012). Some of the vulnerability analysis tools like FinBugs, PMD and Jlint detect the vulnerabilities in Java programs. These tools implement the techniques like static analysis on the byte code of the program. Hence they could not detect some vulnerabilities like zero length array, empty for loop, probable infinite loop etc. In this paper we aim to address this issue. The main contribution of this paper is to provide an

analysis on the Java program vulnerabilities, attacks. To understand their dependencies we use lattice theory based FCA approach. Rest of the paper is organized as follows. Section 2 provides a brief background on FCA. Also it summarizes the literature related to the scope of the current paper. Section 3 provides the model for Java programs vulnerabilities detection. Experiment results are given in Section 4. Analysis of the results is provided in Section 5.

2. Background

2.1 Related work

There is an intensive research in analyzing the source code of the applications for identifying the potential vulnerabilities (Rawat & Saxena, 2009). Though Java language has features with much emphasis on security, programs and applications developed using Java are not free from vulnerabilities. A detailed summary of the Java vulnerabilities can be found in (Long, 2005). Program vulnerability detection and mitigation is well discussed in the literature. A detailed survey on these techniques can be found in (Shahriar & Zulkernine, 2012). Also authors have this article have summarized different patching approaches for mitigating security vulnerabilities and provided a mapping between different mitigation techniques, addressed vulnerabilities and programming languages.

There are several tools to detect the vulnerabilities in the source code. Rutan, Almazan and Foster (2004) have analyzed five different tools for finding vulnerabilities in Java language. Their analysis has revealed that the tools: Bandera, FindBugs, JLint, PMD and ESC/Java2 produces uncorrelated results. Different techniques that are used to develop these tools are summarized in (Shahriar & Zulkernine, 2012). Among the different techniques, static analysis is the most prominent one (Ying et al., 2012). Tools based on the static analysis find the vulnerabilities without executing the code. Parrend (2009) has analyzed three types of classes in the Java components for exploitable vulnerabilities.

FindBugs uses static analysis technique to detect the Java vulnerabilities. It also uses lexical analysis, data flow analysis and syntactic technique to find out the vulnerabilities. It works on analyzing Java class files at bytecode level. However this tool It focuses less on low-level bugs such as memory allocation and dereferencing errors and more on higher level concerns such as API use. PMD detects the Java vulnerabilities with the help of rule set containing more than 140 built- in rules. PMD uses intra-procedural dataflow analysis and lexical analysis techniques. Further PMD detects even duplicate code which is hard to detect. JLint is also an open source tool which performs syntax and semantic analysis to detect the vulnerabilities.

Even though the above mentioned tools can detect many vulnerabilities there are some vulnerabilities that they cannot detect such as zero length array, negative length array, never executed loop, unexpected behavior of the loop etc. These are addressed in the current paper and are discussed in the proposed model.

2.2 Formal concept analysis

Formal concept analysis (FCA) is a method of data analysis with growing popularity across various domains. FCA analyzes data which describe relationship between a particular set of objects and a particular set of attributes in mathematical definition of a formal context. A formal context is a triple $k = (G, M, I)$ where G is a set of objects, M a set of attributes (or items), a binary relation $I \subseteq G \times M$ (Wille, 2005). Two operators $\uparrow$ and $\downarrow$ are defined for every $X \in G$ and $Y \in M$ where

$$X^{\uparrow} = \{m \in M \mid \text{for each } g \in G: (g, m) \in I\} \quad (1)$$

$$Y^{\downarrow} = \{g \in G \mid \text{for each } m \in M: (g, m) \in I\} \quad (2)$$

The two operators $\uparrow$ and $\downarrow$ are called Concept forming operators. The set (X, Y) is called a Formal concept (C). X is called Extension of the concept, and Y is called Intension of the concept. In the standard formal context, the constraint for a formal concept is that all objects in the extent must have all properties in the intent and all properties in the intent must be possessed by all objects in the extent. The concepts can be visualized in a hierarchical order called Concept lattice. Partial order relation that exists between the formal concepts is known as *subconcept-superconcept* relation (Singh & Aswani Kumar, 2012). Interested readers can read extensive introductory information on FCA in (Aswani Kumar & Srinivas, 2010; Kuznetsov & Poelmans, 2013; Priss, 2007).

Association rules can be generated using formal concepts. The extent of the concept is mapped onto the closed frequent itemset and the length of the corresponding intent denotes the support of the itemset. The rules with 100% confidence are called implications. Attribute implications are closely related to functional dependencies in the database field and hence made their way into Association Rules Mining (ARM) problem in data mining (Aswani Kumar, 2011a, 2011b, 2012; Aswani Kumar & Srinivas, 2010a; Li et al., 2013).

Poelmans et al. (2013) have provided an exhaustive review on the FCA applications. Recently FCA has found a prominent role to detect security related trends. Priss (2011) demonstrated the use of FCA for analyzing UNIX system data with respect to IT security monitoring. Breier and Hudec (2012) have used FCA for describing the metrics for security evaluation. Becker et al. (2000) have described a tool that uses FCA methods to model dependencies among security guidelines. Neuhaus and Zimmermann (2009) have used FCA to find the dependencies that increase or decrease the risk of vulnerabilities in

Linux packages. Ganapathy et al. (2007) have used Lindig and Snelting's approach for identifying security sensitive operations in legacy code. FCA is also used to design Role Based Access Control (RBAC) (2013), to model access permissions in RBAC (2012), to model role hierarchy structure in RBAC (Sobieski & Zielinski, 2010). Very recently Sharma et al. (2013) have analyzed the cyber attacks in social dimensions and developed a threat model with FCA as a reasoning engine. In this paper we use FCA to analyze the Java program vulnerabilities and security attacks.

3. Proposed model

Figure 1 shows the architecture of the proposed system. The proposed system receives a Java program as an input from the user. The system checks the existence of the chosen vulnerability in the program. It verifies the vulnerability pattern for its detection. If such vulnerability exists then the system prompts the user about the vulnerability and asks the user whether to verify another vulnerability in that program or in any other program or to exit. Finally detected vulnerabilities in the chosen set of programs can be analyzed using FCA. The proposed model is aimed at detecting the following vulnerabilities:

1. Probable out of bound array indexing vulnerability
   When array length function is used in the initialization or in any condition checking of a loop, then the probability arises to use the value of length of an array as the index inside the loop. This may lead to out of bound array indexing. This case of vulnerability leads to buffer overflow and leads to the following breaches. Due to buffer overflow the Java applets gain elevated privileges. Attackers escape Java sandbox and access arbitrary files via unknown attack vectors.

2. Blocks without braces vulnerability
   This vulnerability occurs when the condition blocks like *if*, *else*, *while*, *try*, *catch* are not within the curly braces. Generally this situation arises when the condition blocks have single line statements. If we consider *if else* blocks the attacker may create a situation such that instead of executing the content inside *if block*, the content in the else block gets executed. This leads to attack called resource tampering attack.

3. Never executed for loop vulnerability
   This vulnerability arises when a programmer declares for loop and leaves it empty. This may lead to cross site scripting attack. The attacker may find out the empty for loop and try to insert malicious code or malicious URL.

4. Unexpected behavior of for loop
   This vulnerability is due to unexpected execution of for loop. This leads to infinite execution of for loop. This type of vulnerability is found using pattern matching.

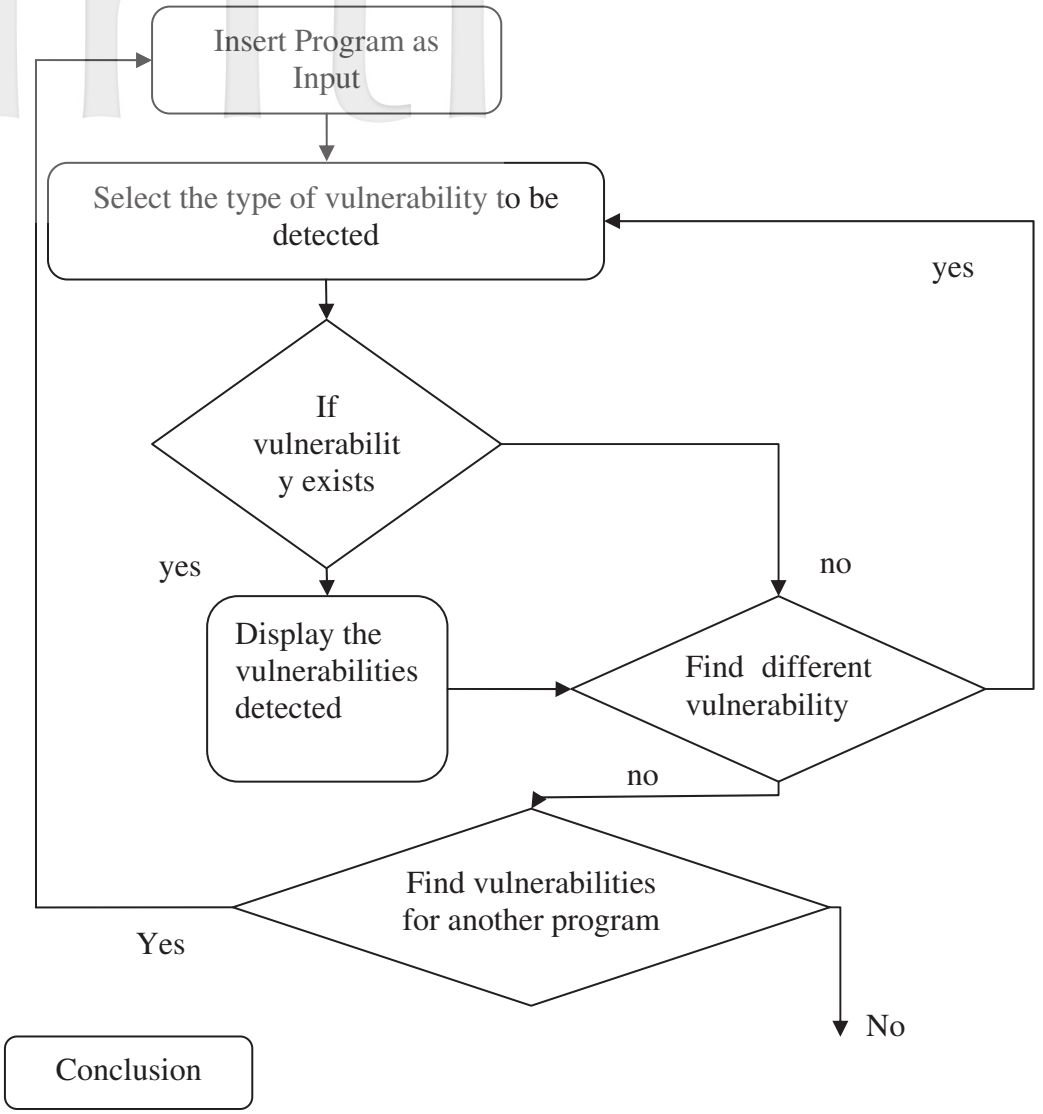

Figure 1 Architecture of the Proposed System

5. Abstract class without abstract method vulnerability
   The abstract class without abstract method vulnerability occurs when the abstract class has no abstract methods in it and may lead to denial of service attacks.

6. Zero length array vulnerability
   The zero length array vulnerability arises when an empty array is declared in the program i.e., an array with zero length size is executed. This leads to unreleased resources attacks.

7. Negative length array vulnerability
   Negative length array vulnerability might occur when the user initialize the array with a negative value or the array length goes to negative value because of some computations in the program. If the array length is negative then it leads to buffer overflow or integer overflow and leads to denial of service attacks.

8. Vulnerabilities related to switch statements
   The vulnerabilities related to switch statements include empty switch statement, too few branches in the switch statement and no default case in the switch statement. If any one of the above three cases are present in the program then the program is vulnerable. The empty switch block allows the attacker to insert some malicious code or URL which contains malicious code.

9. Empty blocks vulnerability
   The empty blocks can be *if*, *while*, *for*, *try*, *catch*, and *finally* loops. The programmer may unintentionally write a loop and leave it empty. This gives the attacker a chance to insert malicious code or data inside the empty loop.

10. Printing exception details
    Printing the exception details is one of the bad practices in Java. Printing the details gives the attacker the information about the exception occurred. Based on the exception details displayed they lead to information disclosure attacks and resource tampering attacks.

11. Unnecessary constructor vulnerability
    The unnecessary constructor vulnerability arises when there is an empty constructor declared in the program. These type of unnecessary constructors results in memory consumption. This kind of vulnerability leads to unreleased resource attacks.

12. Exception related vulnerabilities
    The exception related vulnerabilities include the vulnerabilities like avoid catching null pointer exception, avoid catching throwable exception and avoid throwing null pointer exception.

13. Rethrowing exceptions vulnerability
    Rethrowing an exception is not a good Java practice. Rethrowing the exception again and again leads to denial of service attacks.

14. Unnecessary parenthesis vulnerability
    The unnecessary parenthesis vulnerability is because of the unnecessary use of parenthesis in the return statement. This kind of vulnerability confuses the user and leads to information disclosure attacks.

15. Explicit calling of garbage collector
    This vulnerability occurs when the garbage collector is called explicitly. This kind of vulnerability allows remote attackers to execute arbitrary code.

16. Abrupt program termination vulnerability
    This kind of vulnerability occurs when the user uses the function System.exit(). Access to a function that can shut down the application is an avenue of DOS attacks.

17. Drop table vulnerability
    This vulnerability occurs when a table is dropped from the database abruptly. This kind of vulnerabilities occurs due to SQL statements and leads to SQL injection attacks.

18. Empty array rather than null vulnerability
    This vulnerability occurs when *null* is used in the return statement. When a *null* is returned in the return statement it leads to vulnerability when the client code does not handle *null* properly. This can result in abnormal program termination when the calling function performs operations on *null*. One of the vulnerability that is related to this is the vulnerability in adobe flash. This leads to Denial of Service security attack.

19. Printing absolute path vulnerability
    The function getAbsolutePath() gives the complete path of the file. The path may contain sensitive data which should not be disclosed. The attackers try to get the path and access the files that are not intended to them. This vulnerability leads to path traversal attacks and resource tampering attacks.

20. Or condition in SQL statement vulnerability
    Exploiting the *Or* condition in SQL statements the attackers give the input SQL statement such that the condition mentioned in it is always true.

The system used pattern matching approach to detect the vulnerability. The system verifies each line of the program for chosen vulnerability pattern. For each of the vulnerability chosen for analysis we have defined the pattern based on the type of the vulnerability. The defined patterns are matched against the given Java program. If there is a match then that particular input program is having the vulnerability. In detecting some vulnerabilities string matching method is used in which a particular string is matched with the input. For better understanding we provide some of the vulnerabilities patterns as follows:

- Zero length arrays: This vulnerability is caused when a zero length array is defined or it is passed as an argument.

```
int [] num = new int [0];
```

- Negative length array: This vulnerability is caused when the array is defined with a negative length.

```
int [] zero = new int [-4];
```

- Probable Out of bound array indexing: this vulnerability arises when there is a possibility of out of bound exception in an array.

```
int [] array2 = new int [5];
int b7;
for (int i = 0; i < = array2.length; i++)
{
//statements
}
```

The third statement leads to out of bound array indexing vulnerability.

- Never executed for loop: In some programs there are loops that cannot be executed. Such loops are said to be vulnerable to the attacks.

```
for (int i = 2; i <1; i++)
{
//statements
}
```

The statements inside for loop cannot be executed as the condition in for loop can never be true.

- Unexpected behavior of the loop: some loops in Java program behave unexpectedly because of the condition in the loop. This may lead to infinite execution of the loop.

```
for (int i = 1; i < = 1;i--)
{
//statements
}
for (int i = 2; i > = 2;i++)
{
//statements
}
```

Both for loops lead to infinite execution.

- Braces for if else blocks: some programmers do not use braces for if else blocks when the blocks have single line statements. This may create a problem of resulting in one value instead of other value.

```
int x = 5, y = 3;
if (x = = y)
```

```
if (y = = 3)
x = 3;
else
x = 4;
```

These are the vulnerabilities that are not detected by any of the tools.

4. Experimental results

This section summarizes the experiments we have conducted. We have collected a total of 73 Java programs from different sources. Some of the programs are from applications developed by the students of VIT University as a part of their academic projects. Also some of the programs were taken from web based applications like irctc.co.in, apsrtc.com, vit.ac.in. All these programs are verified for the 20 vulnerabilities listed in the following Table 1. Along with vulnerabilities, Table 1 also summarizes potential attacks that these vulnerabilities cause.

Figure 2 shows the graphical interface of the system that receives the Java program from the user as input and alert the user about the existence of the chosen vulnerability in the code. From the experiments on all the programs we have observed that these programs are either having multiple instances of same vulnerability or multiple vulnerabilities or combination of both. From the experimental results we have observed that the vulnerabilities in application might lead to more than one security attack. Next we analyze the dependencies among the vulnerabilities so as to find occurrence of these vulnerabilities together and similarly attacks. This analysis is performed using FCA.

4.1 Analysis using FCA

In this section we use FCA to analyze the detected vulnerabilities and possible attacks from these vulnerabilities. For this purpose we have created two different formal contexts from the experimental results obtained in the previous section. We have applied FCA on the mapping between the input programs and vulnerabilities detected in each of these programs. For this purpose we have considered the 73 programs as objects (1 to 73) and the 20 vulnerabilities listed above as attributes (a to t) in the formal context. Existence of the vulnerabilities in the programs, detected in previous section, represented the incidence relation between the objects (programs) and the attributes (vulnerabilities). We have used *object intersection* algorithm (Aswani Kumar & Prem Kumar Singh, 2014; Wille, 2005) to find the concepts from this formal context. A total of 69 concepts are generated. Next we have used *Findimplications* algorithm (Aswani Kumar & Prem Kumar Singh, 2014) to find the attribute implications from these concepts. Table 2 summarizes the 26 attribute implication with support count 0 and confidence 100%.

Table 1 List of Vulnerabilities and Attacks Analyzed

S. No	Vulnerabilities Analyzed	Attacks
1	a. Probable out of bound array vulnerability	Denial of service attacks (a)
2	b. Blocks without braces vulnerability	Cross site scripting attacks (b)
3	c. Empty for loops vulnerability	Code injection attacks (c)
4	d. Unexpected loop vulnerability	Information disclosure attacks (d)
5	e. Empty abstract method in an abstract class vulnerability	Path traversal attacks (e)
6	f. Zero length array vulnerability	Resource tampering attacks (f)
7	g. Negative length array vulnerability	Unreleased resources attacks (g)
8	h. Switch branch related vulnerability	SQL injection attacks (h)
9	i. Empty loops vulnerability	
10	j. Printing exception details vulnerability	
11	k. Unnecessary constructor vulnerability	
12	l. Exception related vulnerabilities	
13	m. Re-throwing exceptions vulnerability	
14	n. Unnecessary parenthesis vulnerabilities	
15	o. Explicit calling of garbage collector	
16	p. Abrupt program termination	
17	q. Drop table vulnerability	
18	r. Returning null vulnerability	
19	s. Printing absolute path vulnerability	
20	t. SQL statement or condition vulnerability	

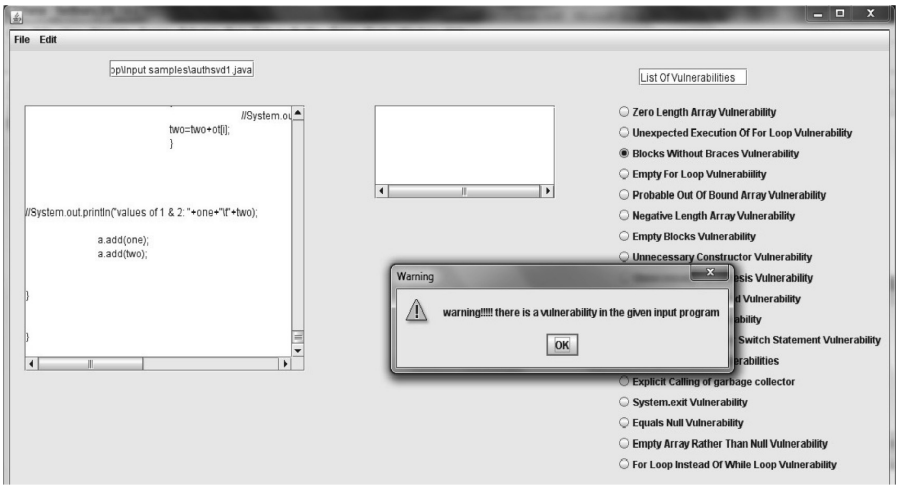

Figure 2 Warning if Vulnerability Exists

Table 2 Rules for Support 0 and Confidence 100%

No	Rules
1	abik → n
2	abjnp → r
3	aij → p
4	bdh → a
5	bik → an
6	bikn → a
7	bjnpr → a
8	bm → n
9	bnp → a
10	c → br
11	cr → b
12	djn → b
13	fk → a
14	hp → b
15	ik → abn
16	ikn → ab
17	in → a
18	jnpr → ab
19	jr → b
20	k → a
21	kn → abi
22	m → bn
23	mn → b
24	npr → abj
25	pr → abjn
26	r → b

These rules summarize the dependencies between the vulnerabilities. The rule abik → n can be interpreted as the programs in which the vulnerabilities set {out of bound array, blocks without braces, empty blocks, unnecessary constructor} exists, then unnecessary parenthesis vulnerability also exists. Also we can observe that no program has all the vulnerabilities in common and no single vulnerability was detected commonly in all the programs.

Next we have analyzed the mapping between the vulnerabilities detected and possible attacks each vulnerability causes. For this purpose, we have constructed a formal

context with detected vulnerabilities as objects (1 to 20) of the context and security attacks as attributes (a to h). Table 3 shows the formal context mapping vulnerabilities and attacks. Table 4 lists the 12 concepts obtained from this context. These concepts are listed as vulnerability (object) and attack (attribute) pairs. Table 5 summarizes 26 implication rules obtained at support count 0 and confidence level 100%. From these rules we can understand that whenever cross site scripting attack occurs, there is a possibility that code injection attack also occurs. Similarly whenever unreleased resource attacks occurs, denial of service attack also occur. We can infer all the other rules similarly. Also this implies the occurrence of the corresponding vulnerabilities jointly together in the programs. We can also observe that no single vulnerability leads to all the security attacks and also no security attack has all the vulnerabilities.

Similarly we have produced are the rules and concepts for support count of 10and confidence level 50%. Table 6 summarizes the rules. It is observed that the cross site scripting attack and code injection attack occur together. Resource tampering attacks

Table 3 Mapping between Vulnerabilities and Security Attacks

	a	b	c	D	e	f	g	h
1	X							
2				X	X	X		
3		X	X					
4	X							
5	X						X	
6	X						X	
7	X						X	
8	X	X	X					
9		X	X					
10				X		X		
11	X						X	
12	X							
13	X							
14	X			X				
15	X							
16	X							
17								X
18	X							
19					X	X		
20								X

Table 4 List of Concepts Obtained

No	Formal Concepts
C1	(ø, a b c d e f g h)
C2	(1 4 5 6 7 8 11 12 13 14 15 16 18, a)
C3	(2, d e f)
C4	(3 8 9, b c)
C5	(5 6 7 11, a g)
C6	(8, a b c)
C7	(2 10, d f)
C8	(14, a d)
C9	(2 10 14, d)
C10	(17 20, h)
C11	(2 19, e f)
C12	(2 10 19, f)
C13	(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20, ø)

Table 5 Rules for Support 0 and Confidence 100%

	Rules
1	ab → c
2	b → c
3	c → b
4	de → f
5	e → f
6	g → a

Table 6 Rules for Support Count 10 and Confidence Level 50%

S. No	Rules
1	b → c
2	c → b
3	d → f
4	e → f
5	f → d
6	f → e
7	g → a

occur whenever path traversal attacks occur but the vice versa is not true. In this case also the unreleased resource attacks leads to denial of service attacks. Table 7 summarizes the rules. We have further generated the concepts and rules for vulnerabilities and security attacks with support count of 10 at confidence level 10%. Among the concepts generated the possibility of the attributes that occur at least 10% in common are considered for rule generation. In this case also it is observed that cross site scripting attacks occur when code injection attacks occur and vice versa. Resource tampering attacks occur whenever path traversal attacks occur but the vice versa is not true. The unreleased resources attacks leads to denial of service attacks with but the vice versa is not true.

Theodoor, Davide and Engin (2012) have mentioned that the occurrence of cross site scripting is correlated with the absence of SQL injection in an application. In the above generated rules and concepts also it is observed that there are no vulnerabilities that are common in both SQL injection and cross site scripting attacks. Similarly no vulnerability leads to both SQL injection and cross site scripting attack. This above observation from the literature (Theodoor et al., 2012) confirms the inference of our analysis.

5. Conclusions

Despite of its security related features, Java has several vulnerabilities. Though there are tools to detect the vulnerabilities in Java programs, they are not successful in detecting several vulnerabilities. This paper has concentrated on detecting such vulnerabilities in Java programs based on their token patterns in the code. Further we have analyzed dependencies in the vulnerabilities using Formal Concept Analysis. Inference of our analysis is in good agreement with the previous observations in the literature.

Table 7 rules for Support Count 10 and Confidence Level 10%

S. No	Rules
1	a → g
2	b → c
3	c → b
4	d → f
5	e → f
6	f → d
7	f → e
8	g → a

Acknowledgements

Authors acknowledge the financial support from National Board of Higher Mathematics, Dept. of Atomic Energy, Govt. of India for the grant number 2/48(11)/21-R&D II/186.

References

Aswani Kumar, Ch. (2011a), 'Knowledge discovery in data using formal concept analysis and random projections', *International Journal of Applied Mathematics and Computer Science*, Vol. 21, No. 4, pp. 745-756.

Aswani Kumar, Ch. (2011b). 'Mining association rules using non-negative matrix factorization and formal concept analysis', *Communications in Computer and Information Science*, Vol. 157, pp. 31-39.

Aswani Kumar, Ch. (2012), 'Modeling access permissions in role based access control using formal concept analysis', in Venugopal, K.R. and Patnaik, L.M. (Eds.), *Wireless Networks and Computational Intelligence*, Springer, Berlin, Germany, pp. 578-583.

Aswani Kumar, Ch. (2012a), 'Fuzzy clustering based formal concept analysis for association rules mining', *Applied Artificial Intelligence*, Vol. 26, No. 3, pp. 274-301.

Aswani Kumar, Ch. (2013), 'Designing role based access control using formal concept analysis', *Security and Communication Networks*, Vol. 6, No. 3, pp. 373-383.

Aswani Kumar, Ch. and Prem Kumar Singh, (2014), 'Knowledge representation using formal concept analysis: a study on concept generation', in Tripathy, B. and Acharjya, D. (Eds.), *Global Trends in Intelligent Computing Research and Development*, Information Science Reference, Hershey, PA, pp. 306-336.

Aswani Kumar, Ch. and Srinivas, S. (2010a), 'Concept lattice reduction using fuzzy k-means clustering', *Expert Systems with Applications*, Vol. 37, No. 3, pp. 2696-2704.

Aswani Kumar, Ch. and Srinivas, S. (2010b), 'Mining associations in health care data using formal concept analysis and singular value decomposition', *Journal of Biological Systems*, Vol. 18, No. 4, pp. 787-807.

Austin, A., Holmgreen, C. and Williams, L. (2013), 'A comparison of the efficiency and effectiveness of vulnerability discovery techniques', *Information and Software Technology*, Vol. 55, No. 7, pp. 1279-1288.

Becker, K., Stumme, G., Wille, R., Wille, U. and Zickwolff, M. (2000), 'Conceptual information systems discussed through an IT-security tool', in Dieng, R. and CorbyKnowledge O. (Eds.), *Engineering and Knowledge Management Methods*, *Models, and Tools*, Springer, New York, NY, pp. 352-365.

Breier, J. and Hudec, L. (2012), 'Towards a security evaluation model based on security metrics,' *Proceedings of 13th International Conference on Computer System and Technologies*, Ruse, Bulgaria, pp. 87-94.

Ganapathy, V., King, D., Jaeger, T. and Jha, S. (2007), 'Mining security-sensitive operations in legacy code using concept analysis,' *Proceedings of the 29th International Conference on Software Engineering*, Minneapolis, MN, pp. 458-467.

Garber, L. (2012), 'Have Java's security issues gotten out of hand?', *IEEE Computer*, Vol. 45, No. 12, pp. 18-21.

Kuznetsov, S.O. and Poelmans, J. (2013), 'Knowledge representation and processing with formal concept analysis', *WIREs Data Mining and Knowledge Discovery*, Vol. 3, No. 3, pp. 200-215.

Li, J., Mei, C., Aswani Kumar, Ch. and Zhang, X. (2013), 'On rule acquisition in decision formal contexts', *Machine Learning and Cybernetics*, Vol. 4, No. 6, pp. 721-731.

Long, F.W. (2005), *Software Vulnerabilities in Java*, Carnegie Mellon University, Pittsburgh, PA.

Neuhaus, S. and Zimmermann, T. (2009), 'The beauty and the beast: vulnerabilities in red hat's packages', *Proceedings of the 29 USENIX Annual Technical Conference*, San Diego, CA, Article No. 30.

Parrend, P. (2009), 'Enhancing automated detection of vulnerabilities in Java components', *Proceedings of International Conference on Availability, Reliability and Security*, Fukuoka, Japan, pp. 216-223.

Poelmans, J., Ignatov, D.I., Kuznetsov, S.O. and Dedene, G. (2013), 'Formal concept analysis in knowledge processing: a survey on applications', *Expert systems with applications*, Vol. 40, No. 16, pp. 6538-6560.

Priss, U. (2007), 'Formal concept analysis in information science,' *Annual Review of Information Science and Technology*, Vol. 40, No. 1, pp. 521-543.

Priss, U. (2011), 'Unix systems monitoring with FCA', *Proceedings. of International Conference on Conceptual Structures*, Derby, UK, pp. 243-256.

Rawat, S. and Saxena, A. (2009) 'Application security code analysis: a step towards software assurance', *International Journal of Information and Computer Security*, Vol. 3, No. 1, pp. 86-110.

Rutan, N., Almazan, C.B. and Foster, J.S. (2004), 'A comparison of bug finding tools in Java,' *Proceedings of 15th International Symposium on Software Reliability Engineering*, Bretagne, France, pp. 245-256.

Shahriar, H. and Zulkernine, M. (2012), 'Mitigating program security vulnerabilities: approaches and challenges', *ACM Computing surveys*, Vol. 44, No.3. doi:10.1145/2187671.2187673

Sharma, A., Gandhi, R., Zhu, Q., Mahoney, W.R. and Sousan, W. (2013), 'A social dimensional cyber threat model with formal concept analysis and fact-proposition inference', *International Journal of Information and Computer Security*, Vol. 5, No 4, pp. 301-333.

Singh, P.K. and Aswani Kumar, Ch. (2012), 'A method for reduction of fuzzy relation in fuzzy formal context', in Balasubramaniam, P. and Uthayakumar, R. (Eds.), *Mathematical Modelling and Scientific Computation*, Springer, Berlin, Germany, pp. 343-350.

Sobieski, S. and Zieliński, B. (2010), 'Modeling role hierarchy structure using the formal concept analysis', *Annales UMCS Informatica*, Vol. 10, No. 2, pp. 143-159.

Theodoor, S., Davide, B. and Engin, K. (2012), 'Have things changed now? An empirical study on input validation vulnerabilities in web applications', *Computers & Security*, Vol. 31, No. 3, pp. 344-356.

Wille, R. (2005), 'Formal concept analysis as mathematical theory of concepts and concept hierarchies' in Ganter, B., Stumme, G. and Wille, R. (Eds.), *Formal Concept Analysis: Foundations and Applications*, Springer, Berlin, Germany, pp. 1-33.

Ying, K., Zhang, Y., Fang, Z., Liu, Q. (2012), 'Static detection of logic vulnerabilities in Java web applications,' *Proceedings of 1th International Conference on Trust*, *Security and Privacy in Computing and Communications*, Liverpool, UK, pp. 1083-1088.

About the authors

Ch. Aswani Kumar is Professor of Network and Information Security Division, School of Information Technology and Engineering, VIT University, Vellore, India. His current research interests are Formal Concept Analysis, Data Mining, Big data, Information Security, and Machine Intelligence. He has published 75 refereed research papers so far in reputed peer reviewed international journals and conferences.

Corresponding author. Network and Information Security Division, School of Information Technology and Engineering, VIT University, Vellore, 632014 India. Tel: +91-416-2202771. E-mail address: cherukuri@acm.org

M. Sai Charitha has received her Master's in Information Technology with specialization in Networking. Her research interests include information security and networking. E-mail address: saicharitha_m@yahoo.com