A new semantic annotation approach for software vulnerability source code

Chi Zhang and Jinfu Chen*

School of Computer Science and Communication Engineering,
Jiangsu University,
Zhenjiang, 212013, China
Email: 2211708035@stmail.ujs.edu.cn
Email: jinfuchen@ujs.edu.cn
*Corresponding author

Lei Zhang

China Information Technology Security Evaluation Center,
Beijing 100085, China
Email: zhangwj8998@126.com

Shujie Chen and Zufa Zhang

School of Computer Science and Communication Engineering,
Jiangsu University,
Zhenjiang, 212013, China
Email: 724486362@qq.com
Email: 1270413566@qq.com

Abstract: An efficient semantic annotation approach is proposed to annotate software vulnerability source code based on the vulnerability code semantic description language (VCSDL) in this paper. A set of general annotation frameworks is proposed for two basic components: basic description information of vulnerability and vulnerability source code description information in the language. Specific annotation methods are studied for these two components, according to the annotation method of the basic description information of vulnerability. Also, the corresponding attribute in the VCSDL document structure is extracted to determine the labelling of the basic information of the vulnerability. While, according to the vulnerability source code information, the semantic annotation of the source code information of the vulnerability is implemented. The experimental results show that the proposed semantic annotation approach has a better effectiveness on the annotation of datasets with a simple code structure and a smaller scale. The success rate and accuracy of the proposed annotation are higher and the false positive rate and false negative rate are lower.

Keywords: software vulnerability; semantic annotation; vulnerability source code; vulnerability detection.

Reference to this paper should be made as follows: Zhang, C., Chen, J., Zhang, L., Chen, S. and Zhang, Z. (2021) 'A new semantic annotation approach for software vulnerability source code', *Int. J. Simulation and Process Modelling*, Vol. 16, No. 1, pp.1–13.

Biographical notes: Chi Zhang is a PhD student in the School of Computer Science and Communication Engineering, Jiangsu University, China. He received his Master's degree from Jiangsu University, China in 2020. His research interests include software testing and service computing.

Jinfu Chen is a Full Professor in the School of Computer Science and Communication Engineering, Jiangsu University, China. He received his PhD in Computer Science and Technology from the Huazhong University of Science and Technology, Wuhan, China in 2009. His major research interests include software testing, software analysis and trusted software. He is a member of the ACM, IEEE CS, and China Computer Federation.

Lei Zhang is an assistant researcher in China Information Security Assessment Center. He received his MA in Software Engineering from the Sichuan University, Chengdu, China in 2010. He received his BE in Electrical Information from the Sichuan University, Chengdu, China in 2007. His research interests include information security and software testing.

Shujie Chen received his BE in Software Engineering from the Jiangsu University, Zhenjiang, China in 2016, where he is currently working hard to the Master's degree in the School of Computer Science and Communication Engineering. His research interests include software testing and service computing.

Zufa Zhang is a Master student in the School of Computer Science and Communication Engineering, Jiangsu University, China. He received his BE in Computer Science from the Jiangsu University, Zhenjiang, China in 2017. His research interests include software testing and service computing.

1 Introduction

Software vulnerability is the software security defect caused by faults in the design and implementation of system hardware and software, or in the formulation and configuration of security policies (Xinhui et al., 2017). These defects may cause some functions of the software to be unrealised or may be illegally used by malicious users or criminals to damage the security of the software and may even destroy the user's computer system (Liu et al., 2013). To address these issues, researchers have developed various anti-virus software and security detection tools to protect against vulnerabilities (Noureddine et al., 2019). With the continuous research on vulnerability information and the rapid development of information security, government departments and some information security companies have also established their own security vulnerability database (Song et al., 2016), but a large number of vulnerability data are currently stored in unstructured ways. In the database, it is not conducive to process and further research on vulnerability information.

Other researches on vulnerabilities tend to focus on vulnerability detection based on the vulnerability source code. Although the current vulnerability detection methods are relatively mature, they do not perform well in terms of detection ability and detection effect. There are still many vulnerabilities that have not been discovered (Chen et al., 2018). In the same programming language, different software security vulnerabilities often show the same features. These features help us to further study the vulnerability on the source code level. Therefore, to analyse the vulnerability pattern at the vulnerability source code level through the analysis of the characteristics of the vulnerability source code, we can use these software vulnerability source codes combined with the current semantic annotation technology. This combination can provide a new vulnerability detection idea and a new method for the study of software security. Therefore, a semantic annotation method for vulnerability source code based on the vulnerability code semantic description language (VCSDL) is proposed in this study. The proposed VCSDL can automatically automate unstructured vulnerability source code files according to the structure of VCSDL. The resulting file enables the translated and uniformly described vulnerability data to be directly processed by the computer, thereby facilitating further research on the vulnerabilities and providing new support for exploiting vulnerability patterns.

The remainder of this paper is organised as follows: The research background is discussed in Section 2. The proposed semantic annotation approach is described in detail in Section 3. The experimental analysis is reported in Section 4. Finally, a conclusion to this paper is provided in Section 5.

2 Research background

There are countless organisations in the world that are studying network security and computer vulnerabilities. Each organisation has different ways of researching on computer vulnerabilities and consequently, the formats of research results recorded are also different (Yan et al., 2016). This has resulted in several challenges such as: finding an effective way to express the details of security vulnerabilities, how to communicate with other organisations and how to browse on a public network. Hence, it would be meaningful to develop a relatively uniform language for describing, communicating and presenting vulnerability information with uniform standards and formats. This language can also be extended to add new vulnerabilities and information at any time. Such a language is known as vulnerability description language (Zarafin et al., 2012).

The research on vulnerability has never stopped from the beginning of vulnerability generation, and the research on vulnerability prevention and detection methods has been ongoing. Researchers have developed a variety of anti-virus software and security detection tools to resist vulnerabilities. Most of the security detection tools will customise the output format specification according to their own data format, so using different security detection tools to detect the same security vulnerability will produce different output format. Especially when there are different naming specifications for these security vulnerabilities, these detection results will be more difficult to associate. Obviously, because of the lack of a unified and effective specification for vulnerability information description, it is difficult for different security detection tools to work together. At the same time, it is difficult for researchers to further study vulnerability information.

VCSDL is a formal vulnerability description language based on extensible markup language (XML) (Garcia-Gonzalez et al., 2017). VCSDL mainly contains the basic description of the vulnerability and the source code of the vulnerability itself. It mainly consists of four elements,

the basic information <Description>, the source code semantic information <AnalysisInfo>, the source code context information <Context> and the vulnerability trigger condition <Condition>. The <Description> is used to describe the basic description of the vulnerability, and the <AnalysisInfo>, <Context> and <Condition> are used to describe the source code of the vulnerability. And each element contains several child elements that describe different aspects of the vulnerability code attribute. The structure of the <Description> element which mainly includes the vulnerability code identification number <ID>, the name of vulnerability <Name>, the common weakness enumeration (CWE) ID of the vulnerability <ID>, the time of the vulnerability mode establishment <Time>, the severity level of vulnerability <Severity>, the type of vulnerability <Category> and other related description information about the vulnerability mode <Summary> is shown in Figure 1. The structure of the <AnalysisInfo> element including the language type of the vulnerability code <Language>, the scope of the analysis <Domain>, the mode of the analysis <Mode> and the sensitivity description <Sensitive> is shown in Figure 2. The structure of the <Context> element including the global variable information <GlobalVar>, the name of the class <ClassName> and the function information <Function> is shown in Figure 3, and each sub-element further includes a plurality of corresponding sub-elements. The structure of the <Condition> element which includes the location of the vulnerability triggered by the source <Line>, the expression of the vulnerability trigger condition <Trigger>, the correlation of the vulnerability trigger condition <Type> and the order of the vulnerability trigger condition <Seq> is shown in Figure 4.

Figure 1 The framework of the Description element

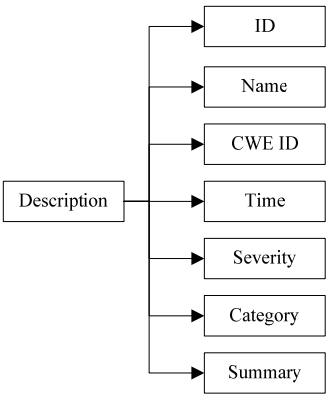


Figure 2 The framework of the AnalysisInfo element

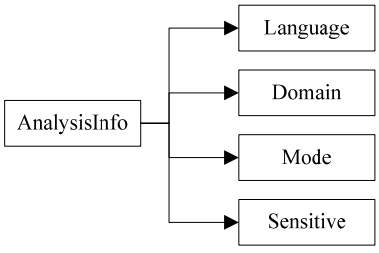


Figure 3 The framework of the Context element

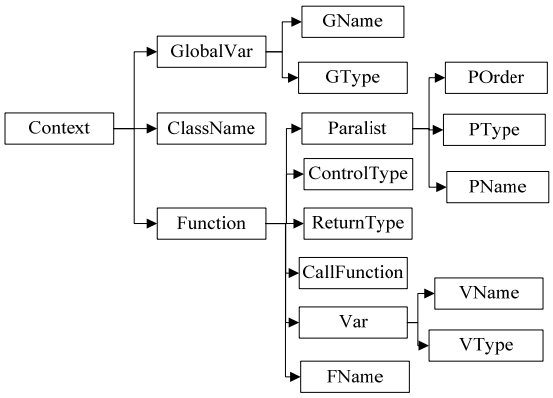
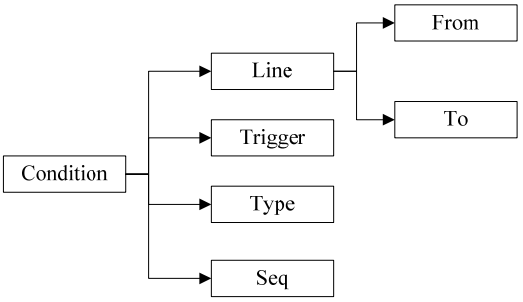


Figure 4 The framework of the Condition element



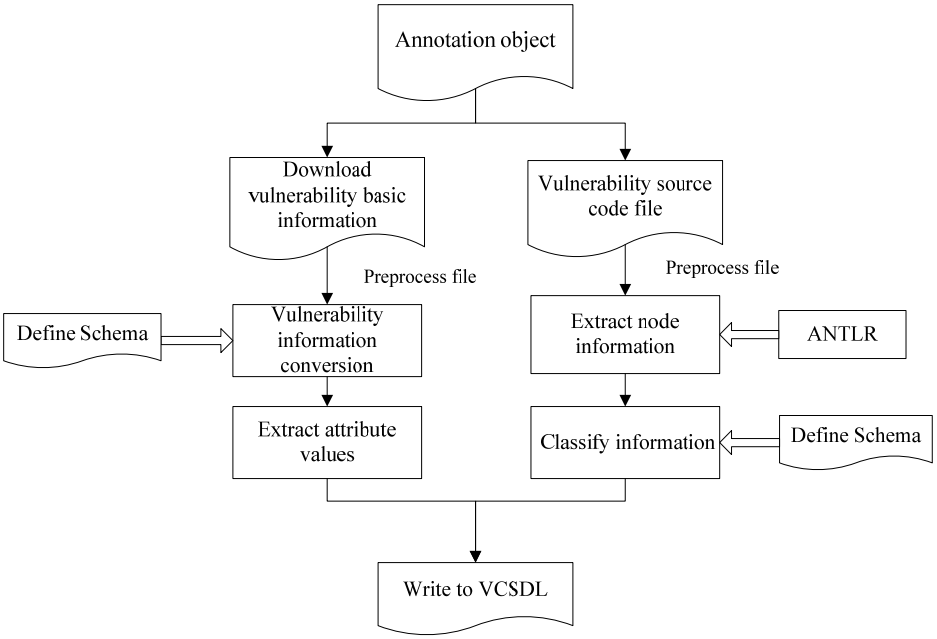
3 The proposed semantic annotation approach

The semantic annotation of the vulnerability source code is to convert the plain text source code file into a structured source file. VCSDL defines a unified vulnerability code description language from the perspective of vulnerability code based on the traditional vulnerability description method (Hu et al., 2009). However, due to the very large size of the existing vulnerability data, it is obviously not advisable to manually convert the vulnerability source code file into a VCSDL file. Therefore, it is necessary to semantically annotate a large vulnerability code set and automate the generation of a VCSDL file.

3.1 General annotation framework

The semantic annotation for vulnerability source code first needs to determine the object to be annotated. A detailed analysis of the vulnerability source code information is performed in the vulnerability database. The current information in the database of the software security vulnerability includes the basic description and the source code of the vulnerability. The basic description of the vulnerability is an important part of the vulnerability information. The basic description information of the vulnerability mainly records the vulnerability number, vulnerability type, danger level, related solutions, etc. The vulnerability source records the source of the vulnerability trigger. The code snippet is a plain text file. Both parts of the vulnerability information contain important descriptive information that is not easily available (Gordan and Dragan, 2019).

Figure 5 General annotation framework for vulnerability source code



The general annotation framework for vulnerability source code based on VCSDL is shown in Figure 5. The general annotation framework of the vulnerability source code is mainly divided into two parts: the annotation of the basic description information of the vulnerability and the annotation of the source code file of the vulnerability. Whereas, the former part initially needs to obtain the basic information of the original vulnerability followed by the pre-processing of this vulnerability file to unify the format of the file and finally extract both the vulnerability data and the required attribute information, and write to the VCSDL file. While the latter part initially needs to obtain the source code file of the vulnerability followed by the pre-processing of the file in order to delete redundant information for facilitating the extraction of information and finally design the corresponding intermediate expression of the program according to the requirements. The symbol table and the abstract syntax tree (AST) are used as an intermediate representation of the program in this paper. The establishment of AST of the vulnerability source code is automatically completed by using the another tool for language recognition (ANTLR) tool and according to the designed symbol table and AST, then traversing AST to complete the extraction of node information to store in the static model. Finally, the node information is classified according to the semantic description elements defined by VCSDL, and the classified node information is written into the VCSDL file.

3.2 Annotation of basic description information of vulnerabilities

The annotation of the basic description information of the vulnerability is mainly aimed to obtain the attribute values of the seven child elements defined by the description element. To get these attribute values, it first needs to get the rawest vulnerability data. The common vulnerabilities and exposures (CVE) and national vulnerability database (NVD) provide downloads of raw vulnerability data. The CVE list page of the CVE official website provides downloadable CVE vulnerability data with many formats, such as XML format, text format, HTML format, comma separated format and so on. In order to facilitate the extraction of the basic information of vulnerability and the implementation of software vulnerability code labelling and query system, it is necessary to establish a database of relational vulnerability to store the basic information of the extracted vulnerability. Therefore, this paper selects vulnerability data with a higher structured XML format. The NVD vulnerability data is downloaded from the data feeds page of the NVD official website's vulnerability menu. The NVD official website stores vulnerability data according to the year classification. It currently includes a total of 18 vulnerability data files from 2002 to 2019. To facilitate the establishment of the basic information database for late vulnerabilities, the vulnerability data files are also downloaded in XML format.

However, the vulnerability data downloaded from CVE and NVD official website is nearly 80,000 in number. This makes it unrealistic to manually convert all these vulnerabilities. Hence, it is necessary to develop and design a method to convert it into batches. Microsoft SQL Server 2017 provides a Business Intelligence Development Studio (BIDS) tool which is mainly used to automatically convert files in XML format into relational databases (Bravo et al., 2016). However, the function of the tool is not fully perfect at present. Only XML files with a simple structure can be effectively converted and XML files with complex namespace cannot be recognised. Therefore, the conversion work cannot be completed. Due to the complex namespace of the vulnerability data files that were downloaded from CVE and NVD in XML format, these original vulnerability

data files need to be pre-processed before the conversion using BIDS tool. Redundant information contained in them is removed and the attribute values required in the <Description> element are retained. A redesign XML Schema without namespace is also needed so that the generated new XML file can be converted and recognised by BIDS. The XML file's partial fragment (nvd-2019.xml) in the original NVD is shown in Figure 6. From the second to the fourth line of the code snippet, it can be seen that the namespace contained in the file is extremely complicated. The BIDS tool cannot recognise the converted file, so the file needs to be pre-processed first.

Figure **6** NVD database XML format fragment

```
<?xml version='1.0' encoding='UTF-8'?>
<nvd
xmlns=http://scap.nist.gov/schema/feed/vulnerability/2.0
xmlns:vuln=http://scap.nist.gov/schema/vulnerability/0.4
xmlns:cvss=http://scap.nist.gov/schema/cvss-v2/0.2 xmlns:
xsi="http://www.w3.org/2001/XMLSchema-instance">
  <entry id="CVE-2019-0001">
    <vuln: cve-id>CVE-2019-0001</vuln:cve-id>
      <vuln:published-datetime>
        2019-01-15T16:29:00.760- 05:00
      </vuln: published-datetime>
      <vuln:last-modified-datetime>
        2019-02-14T13:35:08.587-05:00
      </vuln: last-modified-datetime>
      <vuln: cvss>
      <cvss: base_metrics>
      <cvss: score>7.1</cvss: score>
      <cvss: access-vector>
        NETWORK
      </cvss: access-vector>
      <cvss: access-complexity>
        MEDIUM
      </cvss: access-complexity>
      <cvss: authentication>
        NONE
      </cvss: authentication>
      <cvss: confidentiality-impact>
        NONE
      </cvss: confidentiality-impact>
      <cvss: integrity-impact>
        NONE
      </cvss: integrity-impact>
      <cvss: availability-impact>
        COMPLETE
      </cvss: availability-impact>
      <cvss: source>
        http://nvd.nist.gov
      </cvss: source>
  <cvss: generated-on-datetime>
  2019-01-18T09:20:18.210-05:00
  </cvss: generated-on-datetime>
  </cvss: base_metrics>
  </vuln: cvss>
  <vuln: cwe-id="CWE-400"/>
  ……
```

Taking nvd-2019.xml in the NVD database as an example, the pre-processing of the original file mainly includes the following three steps:

1 Unify the original file format

It is found that the XML file format is not completely unified in the process of analysing the original file. If the attributes in the original file are extracted in batches by using the programming method directly, the program analysis will continue to report errors due to the format problem. This will cause the program to be interrupted and the extraction of attributes cannot be completed. So, the first step is to unify the format of the original file. For example, spaces that appear in files and extra white lines need to be unified first.

The main method used in the unified format is to write the attribute information extracted in the original file into a new file according to a unified format. Its main task is rewriting the seven attribute values defined in the <Description> element to the new format in a unified format. The file re-moves other irrelevant and redundant attribute information. The newly generated file name after the pre-processing is named nvd-2019-prepare.xml, which is shown in Figure 7. It can be seen from the figure, the new XML file contains only the required attribute information, and the format is uniform.

Figure 7 Fragment of nvd-2019-prepare.xml

```
<?xml version='1.0' encoding='UTF-8'?>
 <entry id="CVE-2019-0001">
 <vuln: cve-id>CVE-2019-0001</vuln: cve-id>
 <vuln: published-datetime>
    2019-01-15T16:29:00.760-05:00
 </vuln: published-datetime>
 <vuln: cvss>
   <cvss: base_metrics>
    <cvss: score>7.1</cvss: score>
    <cvss: access-complexity>
      MEDIUM
    </cvss: access-complexity>
   </cvss: base_metrics>
 </vuln: cvss>
 <vuln: cwe id="CWE-400"/>
 <vuln: summary>
    Receipt of a malformed packet on MX Series
  devices with dynamic value configuration can trigger
  an uncontrolled recursion loop in the Broadband Edge
  subscriber management daemon (bbe-smgd), and lead
  to high CPU usage and a crash of the bbe-smgd service.
 </vuln: summary >
 ……
```

2 Design schema for new files

The newly generated schema of the XML file should only contain a specification description of the extracted basic attributes, such as attribute <ID>, <Severity>, <Summary>, etc. The schema should not contain any namespace to facilitate the identification of the BIDS tools. The schema file fragment of the <Description> element is shown in Figure 8.

Figure 8 Schema fragment of the <Description> element

```
<xs:element name="ID" type="xs: string" use="optional"/>
<xs: element name="Name" type="xs: string" use="optional"/>
<xs: element name="CWE ID" type="xs: string" use="optional"/>
<xs: element name="Time" type="xs: date" use="optional"/>
<xs: element name="Severity" type="xs: decimal" use="optional"/>
<xs: element name="Category" type="xs: string" use="optional"/>
<xs: element name="Summary" type="xs: string" use="optional"/>
```

3 Extract the required attribute value information of vulnerability

According to the schema of the design element in the second step, the attribute information of the unified format nvd-2019-prepare.xml is extracted by programming. The main method is to traverse the original file from top to bottom in order by using the file stream operation function in programming. We search for the corresponding data information in the label by performing function matching on different label fields in turn. In order to find the required extracted information and write them into the corresponding label according to the pre-designed schema format, a new XML file is generated and named as nvd-2019-new.xml. The nvd-2019-new.xml fragment is shown in Figure 9. This result is the annotation of the <Description> element in VCSDL.

Figure 9 Fragment of nvd-2019-new.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
  <Vulns>
  <Vuln>
  <Description>
      <ID>VCSDL-400 </ID>
  <CWEID> 400 </CWEID>
  <Name> CVE-2019-0001</Name>
  <Time>2019-01-18</Time>
      <Severity >7.1</Severity>
  <Category> VENDOR_ADVISORY </Category>
   <Summary> Receipt of a malformed packet on MX Series devices with dynamic value configuration can trigger an uncontrolled recursion loop in the Broadband Edge subscriber management daemon (bbe-smgd), and lead to high CPU usage and a crash of the bbe-smgd service.
   </Summary>
  </Description>
  </Vuln>
</Vulns>
```

The above three steps are repeated to complete the conversion of other vulnerability data information in the NVD database. Similarly, the same operation can be performed for the CVE database to generate a new XML file with a simple schema.

The purpose of establishing a relational basic information database of vulnerability is to facilitate the complete annotation of the vulnerability source code in the later stage and the implementation of the software vulnerability source code annotation and query system. For the basic information of the vulnerability source code, the corresponding attribute value can be obtained from the vulnerability basic information base to uniformly generate the VCSDL file in XML format. After pre-processing the original vulnerability file, the BIDS tool in Microsoft SQL Server 2017 is used to perform batch identification conversion of the newly generated XML file.

3.3 Annotation algorithm of the vulnerability source code

The main purpose of the annotation of the vulnerability source code is to generate the VCSDL file in the XML format defined by the <AnalysisInfo> element, the <Context> element and the <Condition> element. The attribute information defined in the AnalysisInfo element and the context element needs to be specifically analysed by the source code of the vulnerability and the condition element describes the trigger condition related information of the vulnerability. Our main task is to extract and label the trigger condition related information. In the VCSDL file, the search for vulnerabilities is not a problem we need to pay attention to, so we can obtain information about the trigger conditions of the vulnerability source code by using the relevant vulnerability static analysis tool. The software vulnerability source code is essentially a program written in a specific language, so the software vulnerability source code has the elements of the programming language such as the class declaration, variable declaration, function definition, data operation, flow control, etc. Therefore, the software vulnerability source code is mainly used for the analysis and information extraction of the vulnerability source code program.

Algorithm 1 shows the VCLBV algorithm. The VCLBV algorithm is based on the VCSDL vulnerability code labelling algorithm. The input of the algorithm is the vulnerability code file set VCF which contains the vulnerability source code file and related reference documents. The output is a VCSDL file in XML format. The main idea is to generate the intermediate expressions corresponding to the vulnerability source code by lexical analysis and grammar analysis of the pre-processed vulnerability source code, namely the symbol table and AST. The following focuses on the design of the symbol table and the AST. Then, through the traversal and filtering of the AST nodes, the node attributes are extracted and a data model is built to store the extracted results. Each logical node in the AST and the defined nodes in the static data model are mapped one by one. The execution time of the algorithm is mainly consumed in the node traversal of AST. Since the total number of nodes traversed does not exceed the total number of nodes of AST, the time complexity of the VCLBV algorithm is O(n) and n is the total number of nodes of the AST.

Algorithm 1 VCLBV algorithm

```
input: Vulnerability source code file set VCF
output: VCSDL file in XML format
Create new storageModel();/*Initialise the static data model */
Create new Node();/*Initialise the AST node set */
preProcessFile(VCF[i]); /* Preprocess file */
reorderFiles();
for(int i=0;i<VCF.count;i++)/* Label each file in turn */
PreLexer lex=new PreLexer(VCF[i]); /* Construct lexical analyser */
TokenStream tokens=new CommonTokenStream(lex);
tokens.toString();/*Lexical analysis of individual files */
Prepaser pp=new Prepaser(tokens);/* Load lexical analysis results into the parser */
  pp.toString ();/*Gramma analysis */
  AST treeRoot=new AST();
  for(int i=0;i<=treeRoot)
    String inf=treeRoot.traversal();//Traversing AST
    if (inf!==null)
        Node.add(inf); /* Extract AST node information into the node set */
    end if;
end for;
for(int i=0;i<=Node.length;i++)
    if(Node(i).match(Schema))
    storageModel.add(Node(i)) /* Match node information to static data model */
VCSDL.write(storageModel);
end for;
return VCSDL;
```

3.4 Design of the symbol table

Usually, the intermediate representation of the source code includes an AST, a symbol table, a data flow graph, a control flow graph, etc. However, the AST and the symbol table are the basis of all other representations. Based on the wide research of the middle of mainstream compilers, this paper firstly designs the symbol table for the source code of the vulnerability. Because the scope of functions or variables in the source code and the nesting relationship between the code are extremely important, this paper designs the symbol table in a tree form and distributes each scope in a separate symbol table. At the same time, in order to satisfy the scope, the most recent nesting principle saves the path from the outermost layer to the current layer by setting a scope stack. The symbol table designed in this paper mainly contains the following four categories:

1 Global symbol table: This table is mainly used to store global scope symbols and symbol related information.

2 Compilation unit symbol table: This table is mainly used to store various symbols and symbol related information in the current compilation unit. Generally, one compilation unit corresponds to one source code file usually.

3 Function local symbol table: This table is mainly used to store the defined symbols and related information within the scope of the function.

4 Block symbol table: This table is mainly used to store the symbols and related information defined in the program block.

The above various symbol tables form a tree hierarchy according to their scoped nesting relationships.

3.5 Concrete design of AST

To facilitate the generation of AST, this paper uses ANTLR (Bau et al., 2010) tool to automatically generate AST. However, the AST structure constructed by ANTLR by default can only provide some basic properties and simple operations and cannot completely display the source code in-formation of the vulnerability. Hence, the structure of the AST needs to be designed according to the mechanism provided by ANTLR (Lu and Xu, 2017). This paper takes the C++ type of vulnerability source code as an example to design the AST. The design of the AST is mainly divided into two parts: the design of the tree node class and the design of the tree structure. Among them, the design of the tree node class is mainly used to save the information of the symbol table and distinguish the different grammatical structures in the program. The tree structure is designed to design the corresponding tree structure according to different grammatical features and traversal requirements. The design of the tree structure is also an important prerequisite for AST generation. The specific design of these two parts is given below.

3.5.1 Design of the node class

It should be noted that the ANTLR constructs the AST node type by default as CommonAST which only contains simple attributes and operations. Thus, to complete the annotation of the source code of the vulnerability, more complete information and operations of the source code are required. This is to enable the node class to be customised to meet the needs.

1 BasicNode class

To facilitate the operation and management of other nodes, a BasicNode class is first customised and used as the parent class of other custom classes, which provides the node's section operations. The specific definition is as follows:

```
class BasicNode: public CommonAST
{
  private:
  string fileName; // Record the file name of the source program where the node is located
```

```
int lineNumber; // Record the line number in the source program where the node is located
public:
virtual void setNodeType (int nodetype); // Set the type of node
virtual void setNodeText (const nodetype::string&txt); // Set the text of the node
virtual void addChild (RefBasicNode c)); // Add child node of AST
virtual BasicNode getFirstChild (); // Get the first child node of AST
virtual BasicNode getNextSib (); // Get the next child node of AST
virtual void setLineNumber (int line); // Set the row of the node in the source program
virtual int getLineNumber () const; // Get the row in the source program where the node is located
virtual void setFileName (const char* fn); // Set the file name of the source program where the node is located
virtual string getFileName () const; // Get the file name of the source program where the node is located
};
```

2 FunctionNode class

The FunctionNode class is the node class corresponding to the virtual node function node. The return type of the function is important information of the function node and it is also the information to be extracted in the annotation process. Therefore, a member variable Symbol *returnType is defined in the FunctionNode class to save the type of the current function return value. It is worth noting that there are virtual functions in the C++ language. Therefore, in order to record whether the current function is a virtual function, you need to add a bool type method IsVirtual() to the class to determine whether the currently parsed function is a virtual function. function.

3 DeclarationNode class

The DeclarationNode class is the node class corresponding to the virtual node declaration node. In the declaration node, the number of declared variables is also important information in the vulnerability source code program. So a member variable VarNum *varNum is defined in the declaration node class to hold the total number of variable declarations in the current declaration node. The declaration of variables in the program takes into account the scope of the problem, so you need to add a method boolisGolbal () to determine whether this variable is a global variable.

4 ClassNode class

The ClassNode class is the node class corresponding to the virtual node class node. Since the design of VCSDL takes into account the description related to the class structure and designs related sub-elements to represent the inheritance relationship between classes and classes, it is necessary to add a variable to save the inheritance relationship between classes in the ClassNode class. <ClassNode*>baseClass.

5 OperatorNode class

The OperatorNode class is the node class corresponding to all the operators that appear in the program. The operator is an overloaded operator of the C++ language. It is used together with other operators to represent an operator overloaded function. Due to this special mechanism, it needs to be in the OperatorNode. Add a member variable OperatorNum *opratorNum to the class to save the number of operations that the overloaded operator has completed.

3.5.2 Design of the tree structure

Since ANTLR can automatically generate AST through the constructed parser, it is necessary to design the AST structure for each of the grammars according to the C++ language in ANTLR and then according to the designed AST structure in the corresponding production. It also involves generating a manual construct that relates to the semantic actions or adding markups, and finally generating a parser that automatically guiding the conversion mechanism of manual ANTLR (Liang et al., 2017). According to the classification standard of grammar in C++ language, C++ statements are mainly divided into function declaration statements, function template statements, loop statements, and selection statements (Kapur et al., 2015). In order to fully convert the required annotation information in the source code of the vulnerability, the tree structure of the corresponding AST is designed for different statements.

1 Function declaration statements

The function declaration statement is a crucial part of the C++ program. When designing the syn-tax tree, the function virtual node in the function statement is used as the root node of the AST, which contains information such as the function name and the return value type of the function. The parameter list node and function body are the child nodes of the root node. In the parameter list node, different parameter nodes are their child nodes and are sibling nodes. In the function body node, different function statements interact with each other as sibling nodes. The grammar description of the AST structure that corresponds to the function declaration statement according to the design description is as follows:

```
function_def:
(functionDecSp)-> ds: functionDecSp)
decName = d:declarator[true]
{AST d2, dc2; d2 = astFactory.dupList(#d); dc2 = astFactory.dupList(#dc);
symbolTable.add(decName, #(null, dc2, d2));
pushScope(decName);}
(declaration)*(VARARGS)?(SEMI!)*{popScope();}
```

```
 compoundStatement[decName]
{##=#(#[NFunctionDef], ##);}
functionDecSp{ int spCount = 0;}:
(options|typeQualifier|('union'|'enum'|'struct|typeSpecifier
[specCount])->spCount=typeSpecifier[spCount])++;
```

2 Function template statements

A function template describes the framework of a function. The function template is also one of the features of the C++ language. When designing the AST of a function template statement, the template is used as the root node of its AST, and the parameter list and function body of the function template are used as the child nodes of the root node which are mutually sibling nodes. The grammar of the AST structure corresponding to the design template function statement is described as follows:

```
template_root :
"template"! LESSTHAN! template_parameterList
GREATERTHAN!
{#template_root=#(#[VIR,"template_def"],#template_root);};
template_parameterList:
{beginTemplateParameterList();}
template_parameter (COMMA! template_parameter)*
{endTemplateParameterList();}
{#template_parameterList=#(#[VIR,"templatelist"]
#templat_parameterList);};
```

3 Loop statements

There are three main types of loop statements in the C++ language which are while loop statements, do-while loop statements and the for-loop statements. Considering the convenience of later maintenance and management, this paper writes the grammar descriptions of these three loop statements in the same grammar description file. The specific grammar design is as follows:

```
iteration_statement!:
    "for" LPAREN
    {enter NewLocalScope();}
        ((declaration) => d:declaration | (e1:expression)?
        SEMICOLON {end_of_stmt();})
        (c: condition)? SEMICOLON {end_of_stmt();}
         (e2: expression)?
RPAREN s: statement
{exitLocalScope();}
{#iteration_statement = #(#[STATEMENT, "for"], #(#[ VIR,
"init"], d, e1), c, #(#[MY, "iterator"], e2), #(#[VIR, "body"],
s));};
| "while" LPAREN
    {enterNewLocalScope();}
        c0: condition RPAREN
        s0: statement
        {exitLocalScope();}
        {#iteration_statement = #（#[STATEMENT, "while"],
        c0, #(#[ VIR, "body"], s0));}
| "do"
    {enter NewLocalScope();}
    s1: statement "while"
    LPAREN e0: expression RPAREN
    {exitLocalScope();}
    SEMICOLON {end_of_stmt ();}
{#iteration_statement = #(#[STATEMENT, "do"],#（#[ VIR,
"body"], s1), #(#[ VIR, "while"], e0) );}
```

As can be seen from the grammar description file of the AST structure corresponding to the loop statement designed in this paper, the description file is mainly composed of three parallel candidates which in turn correspond to for loop statements, while loop statements and do-while loops statement from top to bottom. Among them, the root node of the for-loop statement is set to a for statement. Since the node does not have the actual meaning, a virtual node is used as the root node of the for-loop statement and some symbols information of the for-loop statement are stored in the virtual node. At the same time, the loop variable initialisation statement, the loop condition statement and the loop variable increment statement of the for statement are sequentially used as the child nodes of the root node and the three statements are mutually sibling nodes. For the while loop statement, if its root node is set to while, the actual meaning of the node will be lost. Therefore, a virtual node is used to represent its root node and the conditional statement and execution statement of the while statement are used as children of its root node. They become brother nodes to each other. For the do-while loop statement, if its root node is set to do, the real meaning of the node will be lost. So a virtual node is used to construct the root node; and the execution body and execution condition of the do-while statement are used as child nodes of its root node and they become brother nodes to each other.

4 Selection statements

The main selection statements included in the C++ language are the if selection statement and the switch selection statement. In order to facilitate the later maintenance and management, the grammar descriptions of the two selection statements are written in the same grammar description file. The specific grammar design is as follows:

```
selection_statement:
!"if" LPAREN
    {enter NewLocalScope();}
    condition: condition RPAREN
    left: statement
     options: "else" right: statement)?{exitLocalScope();}
```

```
{#selection_statement = # （#[STATEMENT, “if”],
condition,#(#[EXPRESSION,”left”],left),
#(#[EXPRESSION, “right”], right));}
| “switch”^ LPAREN! {enter NewLocalScope();}
condition RPAREN! statement {exitLocalScope();}
```

According to the grammar description file, *if* is the root node for the *if selection* statement, but the actual meaning of the node is lost, so a virtual node STATEMENT is defined as its root node, and the conditional statement and the execution statement are used as its child nodes. The execution statement includes an *if* statement and an else statement, which are sibling nodes. It can be seen from the grammar description file that the grammar production of the *switch* statement is much simpler than the *if* statement. The root node of the AST corresponding to the *switch* statement is switch; and the conditional statement and the execution body are child nodes, which are mutually sibling nodes. At the same time, the case node and the default node are the child nodes of the execution body node, and they are also sibling nodes.

3.6 Design of traversal method of AST

All node information of the AST can be obtained by sequentially traversing each node in the AST generated by the vulnerability source code file and then the attribute information defined in the <AnalysisInfo> element and the <Context> element is extracted by filtering the node information. The node traversal of AST can be easily complete using AST traversal mechanism of ANTLR, but in the process of traversal, the traversal operations involved in the different node classes designed in Section 2.5 are different. Distributing these traversal operations in different node classes can result in reduced traversal efficiency and recompilation of these node classes if new traversal operations need to be added. In order to improve the efficiency of traversal and facilitate post-maintenance, this paper considers the use of the visitor pattern to separately encapsulate the related operations of different node classes into a single object. When traversing the AST, the operation object is firstly passed into the node that is currently traversed. Then when the current node receives the object, it will send a request containing the information of its own node. At the same time, the request also treats the current node itself as a parameter and finally the corresponding traversal operation for the node. The following four roles are designed based on the role of the visitor pattern combined with the AST traversal process:

1 Abstract visitor role: This role is mainly used to declare the access operation interface. The interface provides an access method visit() for different node classes. This method is mainly used to accept the corresponding node object as a parameter to provide access operations.

2 Specific visitor role: This role is mainly used to implement the interface defined in the abstract visitor role. It is the access operation of different function types defined in the abstract visitor.

3 Abstract node role: This role mainly defines a receiving method accept() which is used to receive a visitor object as a parameter. At the same time, the role also defines a childrenAccept() method which takes a specific node object as a parameter to get all the child nodes of the node.

4 Specific node role: This role is mainly used to implement the methods defined in the abstract node and implement different operations according to different node types.

The traversal of the AST firstly needs to deal with the call relationship between the functions, find the function called by the function and the function that calls the function by traversing the function list. If there is no call relationship, a virtual node is created to represent the function node that does not exist in the current source code, that is, there is no function call relationship. Then, the current node data structure, function definition, function declaration, variable declaration and other phase relationship information are sequentially identified and extracted and then the child nodes information of the current node is recursively identified. The information of the extracted vulnerability code attribute is stored in a data model for reading and writing into the XML file. The correspondence between the logical node in the AST and the node in the static data model is shown in Table 1.

The attributes in the static model contain the basic information of the source code structure of the vulnerability, that is, the annotations of the <AnalysisInfo> element and the <Context> element are completed. For the related attributes of the vulnerability trigger condition described in the <Condition> element, this paper selects the relevant static according to different language types. The analysis method performs static analysis on the source code of the vulnerability. For example, for the C++ language, there are some commonly used static analysis methods such as FlawFinder, SPLINT, CppCheck, etc. For the Java language, there are some commonly used methods such as FindBugs, FindSecBugs, PMD, etc. These static analysis methods have been tested by the project team are known to have strong abilities in detecting vulnerabilities. Through the monitoring and analysis of the mining state, the number of vulnerabilities in the vulnerability file, the cause of the vulnerability and the location of the vulnerability triggering can be obtained, which satisfies the description of the vulnerability triggering condition related attributes in the <Condition> element. Finally, the attribute information in the static model is classified and corresponding to each element in the VCSDL, and the XML file in the VCSDL format can be generated by using the XML related programming or the XML generation tool.

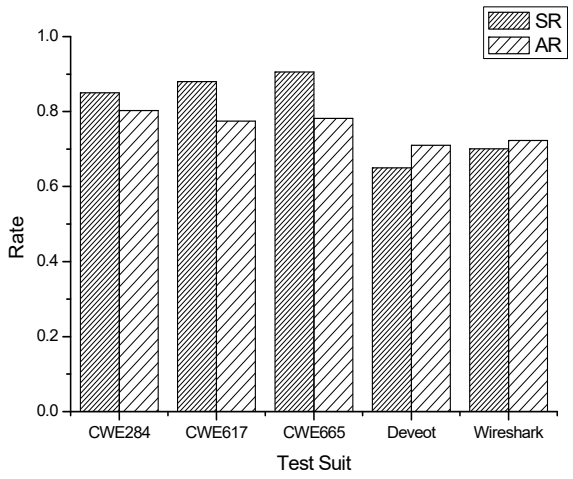
Table 1 Corresponding relationship of node

Source code logical structure node type		*Static model*	
file	File	SourceFile	Source code file
function_definition	Function definition	Function	Function definition
compound_statment	Block	Block	Block
funCall	Function call	FunCallExp	Call expression
assignment	Assignment expression	AssignmentExp	Assignment expression
typedef	Typedef definition	TypeDefine	Typedef definition
init_declarator_list	Variable declaration	VarDeclation	Variable declaration

4 Experiments and analysis

To verify the feasibility and effectiveness of the semantic annotation method for vulnerability source code proposed based on VCSDL in this paper, Juliet Test Suite and complete open source projects were chosen as the vulnerability dataset for the experiment. The experiment used Juliet Test Suite for C/C++ version 1.2, from which CWE284_Improper_Access_Control, CWE617_Reachable_Assertion, CWE665_Improper_Initialisation were selected. Deveot and Wireshark are open source projects with loopholes that were selected from GitHub. The vulnerability dataset is preprocessed, and the details of the processed vulnerability dataset are shown in Table 2.

Figure 10 Success rate and accuracy of labelling for five vulnerability datasets



In the experimental result of Table 3 and Figure 10, the total number of files in the vulnerability dataset is recorded as FN, the total number of tags in all generated VCSDL files in XML format is recorded as LN, and the number of valid tags that are successfully generated is marked as ELN. In addition, the total number of correct labels obtained by human comparison analysis is recorded as RLN and the time cost is recorded as T. By definition, the annotation of success rate SR = ELN / LN and the labelling accuracy rate AR = RLN / LN. Obviously, the higher the success rate and accuracy, the lower the time overhead, which proves that the method is more feasible and has better performance. The results of proposed annotation for the five vulnerability datasets are shown in Table 3, and the histogram of the success rate and accuracy of the annotation are shown in Figure 10. As can be seen from Table 3 and Figure 10, for the Juliet vulnerability dataset with small code size and simple structure, the success rate and accuracy of the annotation are high. For Deveot and Wireshark, which have larger code sizes and relatively complex code structures, the success rate and the labelling accuracy are reduced to some degree.

Figure 11 False positive rate of three vulnerability datasets to be detected by six methods (see online version for colours)

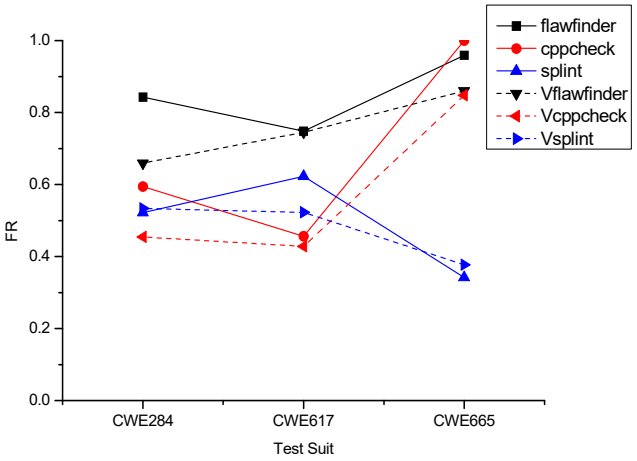
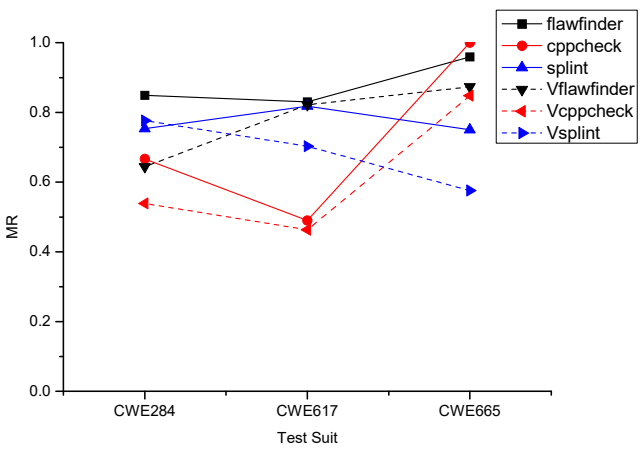


Figure 12 False negative rate of three vulnerability datasets to be detected by six methods (see online version for colours)



The experimental results are shown in Table 4, Figures 11 and 12, which show the total number of vulnerabilities detected in the datasets that were recorded as FN and the total number of vulnerabilities detected by the method is recorded as RN, the number of correct vulnerabilities detected by the method is recorded as TP, and the number of false vulnerabilities detected by the method are recorded as FN. Obviously, FN = RN – TP. The false positive rate and false negative rate are recorded as FPR and FNR respectively. According to the definition, FPR = FN / RN, FNR = (VN – TP) / VN. The lower the false positive rate and false negative rate of a detection method, the better the detection effect. In the experimental results, Vflowfinder, Vcppcheck and Vsplint, respectively represent the methods of using vulnerability code labelling method based on VCSDL (VCLMBV) to detect vulnerabilities. The detection effectiveness of different detection methods is compared by comparing the false positive rate and false negative rate. The detection results of six detection methods for three vulnerability datasets are shown in Table 4. The line charts of false positive rate and false negative rate are shown in Figure 11 and Figure 12. The false positive rate and false negative rate of the three vulnerability data sets are reduced to some extent after using VCLMBV as shown in Figure 11 and Figure 12.

Table 2 Information of the experimental vulnerability dataset

Vulnerability dataset	*Number of files*	*Total number of lines of code*	*Total number of vulnerabilities*
CWE284_Improper_Access_Control	219	28,689	219
CWE617_Reachable_Assertion	488	68,320	488
CWE665_Improper_Initialisation	316	50,876	316
Deveot	811	147,220	811
Wireshark	2,281	1,625,396	2,281

Table 3 Experimental result

Vulnerability dataset	*FN*	*LN*	*ELN*	*RLN*	*SR*	*AR*	*T(s)*
CWE284_Improper_Access_Control	219	3,723	3,165	2,540	85.01%	80.25%	1,320
CWE617_Reachable_Assertion	488	9,760	8,590	6,652	88.01%	77.44%	2,451
CWE665_Improper_Initialisation	316	5,688	5,152	4,028	90.58%	78.18%	2,535
Deveot	811	11,304	7,345	5,214	64.98%	70.99%	8,630
Wireshark	2,281	20,978	14,695	10,625	70.05%	72.3%	20,618

Table 4 Test results of six analysis methods

Vulnerability dataset	*VN*	*Test methods*	*RN*	*TP*	*FN*	*FPR*	*FNR*
CWE284_Improper_Access_Control	219	flawfinder	210	33	177	84.29%	84.93%
		cppcheck	180	73	107	59.44%	66.67%
		splint	113	54	59	52.21%	75.34%
		Vflawfinder	229	78	151	65.94%	64.38%
		Vcppcheck	185	101	84	45.41%	53.88%
		Vsplint	105	49	56	53.33%	77.62%
CWE617_Reachable_Assertion	488	flawfinder	330	83	247	74.85%	82.99%
		cppcheck	458	249	209	45.63%	48.98%
		splint	236	89	488	62.29%	81.76%
		Vflawfinder	341	87	254	74.49%	82.17%
		Vcppcheck	458	262	196	42.79%	46.31%
		Vsplint	304	145	159	52.3%	70.29%
CWE665_Improper_Initialisation	316	flawfinder	316	13	303	95.89%	95.89%
		cppcheck	316	0	136	100%	100%
		splint	120	79	41	34.17%	75%
		Vflawfinder	286	40	246	86.01%	87.34%
		Vcppcheck	316	48	268	84.81%	84.81%
		Vsplint	215	134	81	37.67%	57.59%

In general, for the vulnerability dataset with simple code structure and small scale, the semantic annotation method based on VCSDL can provide a more efficient annotation for vulnerability source code. However, for the vulnerability dataset with larger code size and more complex structure, the overall efficiency of the method is slightly affected, but its overall impact is small.

5 Conclusions

In this paper, a semantic annotation approach is proposed to annotate software vulnerability source code based on VCSDL, and a specific annotation method is formed for the two components of vulnerability basic description information and vulnerability source code description information in the language. The experimental result shows that the proposed method can effectively transform the unstructured vulnerability source code file into a structured tagging result file. Also, it has high labelling efficiency, and lower false positive rate and false negative rate for the vulnerability dataset with simple code structure and small scale. The efficiency of labelling vulnerability code with complex program code and a large amount of code will be affected to some extent, hence the labelling method proposed in this paper needs to be further improved and optimised based on multi-label algorithm in future.

Acknowledgements

This study was funded by the National Natural Science Foundation of China (Nos. U1836116), the Project of Jiangsu Provincial Six Talent Peaks (No. XYDXXJS-016), and the Graduate Research Innovation Project of Jiangsu Province (No. KYCX17 1807).

References

Bau, J. et al. (2010) 'State of the art: automated black-box web application vulnerability testing', *2010 IEEE Symposium on Security and Privacy.* , IEEE, pp.332–345.

Bravo, M., Rodriguez, J. and Pascual, J. (2016) 'SDWS: semantic description of web services', *International Journal of Web Services Research*, Vol. 11, No. 2, pp.1–23.

Chen, Z., Zhang, Y. and Chen, Z. (2018) 'A categorization framework for common computer vulnerabilities and exposures', *Computer Journal*, Vol. 53, No. 5, pp.551–580.

Garcia-Gonzalez, H., Gayo, L.J.E. and Paule-Ruiz, M. (2017) 'Enhancing e-learning content by using semantic web technologies', *IEEE Transactions on Learning Technologies*, Vol. 10, No. 99, pp.544–550.

Gordan, T. and Dragan, J. (2019) 'Modelling, simulation and resource optimissation of complex development project by fusion of multiple-domain matrix and coloured Petri nets methods', *International Journal of Simulation & Process Modelling*, Vol. 14, No. 1, pp.51–63.

Hu, L., Ying, S., Zhao, K. et al. (2009) *2009 WASE International Conference on In-formation Engineering – A Semantic Web Service Description Language*, pp.449–452, [*IEEE 2009 WASE International Conference on Information Engineering (ICIE)* – Taiyuan, Shanxi, China (2009.07.10-2009.07.11)].

Kapur, P., Yadavali, V.S. and Shrivastava, A. (2015) 'A comparative study of vulnerability discovery modeling and software reliability growth modeling', *Futuristic Trends on Computational Analysis and Knowledge Management*, IEEE, pp.246–251.

Liang, H., Liu, S., Zhang, Y. et al. (2017) 'Improving the precision of static analysis: Symbolic execution based on GCC abstract syntax tree', *IEEE/ACIS International Conference on Software Engineering*, IEEE, pp.395–400.

Liu, B., Shi, L., Cai, Z et al. (2013) 'Software vulnerability discovery techniques: a survey', *Fourth International Conference on Multimedia Information Networking & Security*, IEEE, pp.152–156.

Lu, Y. and Xu, X. (2017) 'A semantic web-based framework for service composition in a cloud manufacturing environment', *Journal of Manufacturing Systems*, Vol. 42, No. 1, pp.69–81.

Noureddine, G., Abdelkrim, A., Mourad, O. et al. (2019) 'Developing an evolution software architecture framework based on six dimensions', *International Journal of Simulation & Process Modelling*, Vol. 14, No. 4, pp.325–337.

Song, J.H., Park, J.P. and Jun, M.S. (2016) 'A study of vulnerability assessment using fuzzing data suite and data flow analysis in software', *Advanced Science Letters*, Vol. 22, No. 9, pp.2592–2597.

Xinhui, H., Shuang, W., Jiayi, Y.E. et al. (2017) 'Detect use-after-free vulnerabilities in binaries', *Journal of Tsinghua University (Science and Technology)*, Vol. 57, No. 10, pp.1022–1029.

Yan, J., He, H., Zhong, X. et al. (2017) 'Q-learning-based vulnerability analysis of smart grid against sequential topology attacks', *IEEE Transactions on Information Forensics & Security*, Vol. 12, No. 1, pp.200–210.

Zarafin, A.M.A., Zimmermann, A. and Boissier, O. (2012) 'Integrating semantic web technologies and multi-agent systems: a semantic description of multi-agent organizations', *AT2012*, pp.1–2.