Ontology Model-Based Static Analysis of Security Vulnerabilities

Lian Yu[1], Shi-Zhong Wu[2], Tao Guo[2], Guo-Wei Dong[2], Cheng-Cheng Wan[1], and Yin-Hang Jing[1]

[1] School of Software and Microelectronics, Peking University, Beijing 102600, China
[2] China Information Technology Security Evaluation Center, Beijing 100085, China
lianyu@ss.pku.edu.cn, {tiger,guotao,donggw}@itsec.gov.cn, sitang@pku.edu.cn

Abstract. Static analysis technologies and tools have been widely adopted in detecting software bugs and vulnerabilities. However, traditional approaches have their limitations on extensibility and reusability due to their methodologies, and are unsuitable to describe subtle vulnerabilities under complex and unaccountable contexts. This paper proposes an approach of static analysis based on ontology model enhanced by program slicing technology for detecting software vulnerabilities. We use Ontology Web Language (OWL) to model the source code and Semantic Web Rule Language (SWRL) to describe the bug and vulnerability patterns. Program slicing criteria can be automatically extracted from the SWRL rules and adopted to slice the source code. A prototype of security vulnerability detection (SVD) tool is developed to show the validity of the proposed approach.

Keywords: Static analysis, Program slicing, Vulnerability ontology model, Reasoning.

1 Introduction

Software security has become a matter of paramount concerns in many kinds of modern organizations today. This trend has motivated considerable researches in software security assurance. At present, there are many possible techniques that can be applied to detect software vulnerabilities and to improve software security, such as code reviews, inspections, testing technology, runtime verification, and static analysis. Among these, static analysis is particularly well suitable to security because many security problems occur in corner cases and hard-to-reach states that can be difficult to exercise by actually running the code. Good static analysis tools provide a fast way to get a consistent and detailed evaluation of a body of code.

In recent years, many static analysis tools have been developed to find bugs in software. For example, the ESC/Java (http://kind.ucd.ie/products/opensource/ESCJava2/), FindBugs (http://findbugs.sourceforge.net/), JLint (http://artho.com/jlint)

S. Qing et al. (Eds.): ICICS 2011, LNCS 7043, pp. 330–344, 2011.
© Springer-Verlag Berlin Heidelberg 2011

and PMD (http://pmd.sourceforge.net) are all bug finding tools. The general process of static analysis performed by these tools can usually be divided into three steps:

1. Build the models of the source code by parsing the source code file.
2. Collect bug & vulnerability patterns and develop models for them. Different static analysis tools have different representation of bug pattern models, for example, PMD directly writes the bug patterns as rules in the form of Java classes and checks the rules while traversing the AST of the source code to find bugs.
3. Check the defects in the program models derived in the first step against the rules developed in the second step, and then produce the bug and vulnerabilities reports.

There are a number of limitations in these traditional static analysis approaches.

- Most of the bug patterns are for general classes of software defects rather than software security vulnerabilities. Moreover, bug pattern models are tightly coupled with the tools' implementation, thus extending new bug patterns applicable with a specific context will be difficult to perform, since people need to get involved in the great details of the programmatic implementation. Although some tools, such as ESC/Java2, provide means to write custom detectors, the extensibility is still limited by the design and capability of the analysis tools.
- There is a lack of formal and expressive mathematical language to define and describe the security vulnerabilities and bug patterns. Informal definition of software vulnerabilities and bugs involuntarily leads to ambiguous understanding about software defects and will potentially bring up false positives which can be time-consuming to address. Moreover, traditional bug categorization is not sufficient to describe the complex concepts in a specific context and the relationships that hold between those concepts, thus potentially it brings up false negatives which hide deeply within the system but may cause big problems under specific environments.
- The detecting approaches based on certain programming language models differ as the language changes. The models and analysis modules are hard to reuse, so it will be costly if the static analysis tools want to support various kinds of programming languages. Implementation of similar security rules in different languages may differ greatly, thus the whole system is unable to reusable.

To address these problems, we propose an ontology-model based static analysis approach to detect software vulnerabilities. We have put forward a preliminary idea of using a similar approach to automatically detect bugs in the source code of Java programs in the paper [1] and have developed a tool to verify its effectiveness. However, the greatest challenge at the moment adopting the approach is the enormous scale of program models the approach may generate when dealing with large software systems. The large scale problem could reduce the performance of the whole reasoning system to a large extent. This paper goes a step further to discuss the possibility of combining the ontology-based approach with traditional static analysis technologies such as program slicing, which we will describe later in this paper, to

address the problem and improve the practical utility of the methodology. This paper uses Ontology Web Language (OWL) to model the source codes and Semantic Web Rule Language (SWRL) to describe the bug and vulnerability patterns. Program slicing criterion can be automatically extracted from the SWRL rules and adopted to slice the source code. Program elements that are irrelevant to those criteria will be sliced off and will not be considered in the following phases. The adoption of program slicing can greatly improve the performance of the security vulnerability detection tool.

The rest of the paper will be organized as follows. We first present and discuss the improved overall process of the analysis approach in Section 2. We describe the details of modeling programming language and vulnerability rules in Section 3, and the program slicing solution in Section 4. We carry out experiments in Section 5 and review related work in Section 6, and finally we conclude the paper and scratch the future work in Section 7.

2 Overall Process

In this section we discuss the overall process of the program slicing-enhanced ontology model-based static analysis approach.

The approach follows the three steps which are similar to the general approach of static analysis but in different ways: modeling Java specification and bug patterns using ontology techniques; modeling source code into an abstract syntax tree (AST) and then converting to ontology individuals, where we use program slicing technology in this step to reduce the number of individuals and so as to improve the efficiency of the following reasoning task; and reasoning and generating bug reports. A more detailed process consisting of three phases is shown in Figure 1.

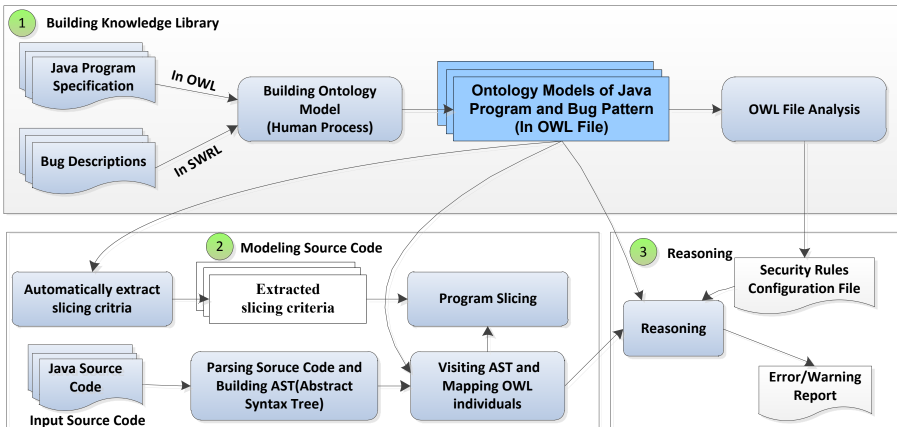

Fig. 1. Process of Ontology Model-Based Static Analysis

1. The first phase, called "Building Knowledge Library", consists of the following two steps.
   a. Modeling Java programming language specifications into ontology models in OWL. Most traditional approaches won't have this modeling step, they usually directly implements language models in specific data structures and program modules. We will illustrate this phase in details in Section 3.
   b. Describing bug patterns and vulnerabilities with SWRL (Semantic Web Rule Language). The purpose of this step is to convert the bug and vulnerability patterns to internal representations that are suitable for a particular analysis. There are many vulnerability listings and publications such as the Common Vulnerabilities and Exposures (CVE) list(http://cve.mitre.org/), CWE(http://cwe.mitre.org/), CERT (http://www.cert.org/nav/index_red.html/), SANS(http://www.sans. org/), and the Open Web Application Security Project (OWASP), all of which can be the sources of the bug and vulnerability patterns.
2. The second phase, called "Modeling Source Code", can be divided into four steps.
   a. Parsing the source code and converting to an AST.
   b. Constructing the Procedure Dependence Graphs (PDGs) for the procedures (methods) and the System Dependence Graph (SDG) of the entire system.
   c. Slicing over the dependence graphs using the criteria automatically extracted from the SWRL rules. The adoption of program slicing technique is to reduce the scale of the program to be detected.
   d. Mapping the sliced nodes to OWL individuals. Based on the AST, a module that can traverse the AST will then map the nodes to OWL individuals, which are the target outputs of the entire second phase and thus the "internal representations" of the source code.
3. The third phase is reasoning. The purpose of this step is to analyze the internal representations of the source code so as to detect vulnerabilities and generate reports. In our approach this task is performed by a reasoning engine. We will discuss this step in Section 4.4.

3 Modeling Language Specification and Bug Patterns Using Ontology Techniques

This section discusses how to model programming language specification and vulnerability patterns with ontology techniques. We will take the Java language as a case study.

3.1 Modeling Java Specification

In computer science and information science, ontology is a formal representation of knowledge as a set of concepts within a domain, and the relationships between those concepts.

OWL is a famous ontology language for making ontology statements and it's a W3C standard. OWL is developed based on RDF (Resource Description Framework) and RDFS (RDF Schema). All of its elements (classes, properties and individuals) are defined as RDF resources, and are identified by URIs. In our approach we choose OWL to model the Java specification.

Thing in OWL domain is similar to the java.lang.Object in Java, as all OWL classes are Thing's descendants, while all Java classes are descendants of the java.lang.Object class. Both Java and OWL use the term Class to describe concepts in a domain. Class instances represent individuals in OWL and objects in Java. OWL defines some built-ins (properties) to describe relationships of concepts. These properties can also be used to describe the relationships of Java elements. For example, the "owl:subclass-of" property can be used to describe the inheritance relationship of Java classes.

As a formal language representation, BNF (Backus-Naur Form) production is the most widely used form of language lexical and syntax definition. Many tools such as JavaCC (Jhttp://javacc.java.net/) and ANTLR (http://www.antlr.org/) use BNF to describe the Java grammars. We also use the Java BNF in our modeling phase. We use OWL Class to describe all elements of Java language, and use OWL properties to describe the relationships between these elements.

Totally, we define 98 OWL classes to cover all the Java elements. We give part of the OWL classes and properties together with their corresponding Java explanations in Table 1 and Table 2. For more detailed information, see our previous work in paper [1].

Table 1. Relationship between Classes in OWL and Java Elements

OWL Classes	Elements in Java Specification
ClassDesc	Class in Java
InterfaceDesc	Interface in Java
MethodDesc	user defined methods
AttributeDesc	The fields in class or interface file
ObjectDesc	The object variable used in Java source code
ModifierDesc	Modifier of class, interface method attribute, such as: public, private, protected

Table 2. OWL properties and corresponding relationships in Java

OWL Properties	Relationships Between Java Elements
ClassDesc_hasExtends	Inheritance relationship between Java classes
ClassDesc_hasMehtod	Each class may have a few methods
ClassDesc_hasAttribute	Each class may have a few attributes
ClassDesc_hasModifier	Each class must have a modifier, most common it's
ObjectDesc_hasType	Each object must belong to some type(class)
AttributeDesc_hasType	Attribute also has types

3.2 Modeling Bug and Vulnerability Patterns

We collect more than 200 Java bug patterns from OWASP, CWE, and Jtest and convert them into SWRL rules. In this section, we will illustrate how to construct SWRL rules from bug patterns.

Take the TRS.CSTART-3 rule from Jtest which indicates that "Do not call the start method of threads from inside a constructor" as an example. We write the following SWRL rule (see Figure 2) which describes "A class with any constructor within which a thread object's start method is invoked will infer an error report".

```
SWRL Rule
ClassDesc(?className) ∧ClassDesc_hasMethod(?className, ?methodName) ∧MethodDesc(?methodName) ∧
name(?className, ?classLableName) ∧name(?methodName, ?methodLableName) ∧swrlb:equal(?classLableName, ?methodLableName) ∧
MethodDesc_hasClause(?methodName, ?c) ∧InvokeClause(?c) ∧InvokeClause_invokedObject(?c, ?o) ∧Object_hasType(?o, ?n) ∧
name(?n, "Thread") ∧InvokeClause_invokedMethod(?c, ?m) ∧
name(?m, "start") ∧NotCallStartInConstructMethodWarning(?w)
 → errorReport(?c, ?w)
```

Fig. 2. The detection rule definition in SWRL

SWRL rules are critical for the reasoning process because it directly influences the results. If we adopt the rule in Figure 2 to detect the following codes as shown in Figure 3, it will generate an error report since the codes violate the rule. However, this is a typical false positive caused by the ambiguous understanding about the software defect.

```java
package basicbugs;
import java.util.List;

public class CallStartInConstructMethod {
    private List<Thread> workingList;
    public CallStartInConstructMethod(){
        if(workingList != null){
            for(Thread e:workingList){
                e.start();
            }
        }
    }
}
```

Fig. 3. False positives caused by the imprecise rule

The precise definition of this rule, which should be "Code where 'start()' is invoked inside the constructors of classes that extend java.lang.Thread will be considered as an error", is shown in Figure 4. Note the added conditions in the antecedent in the rule are marked with red box.

```
SWRL Rule
ClassDesc(?className) ∧ClassDesc_hasMethod(?className, ?methodName) ∧MethodDesc(?methodName) ∧
ClassDesc_hasExtend(?className,?extend_c) ∧ name(?extend_c, "Thread") ∧
name(?className, ?classLableName) ∧name(?methodName, ?methodLableName) ∧swrlb:equal(?classLableName, ?methodLableName) ∧
MethodDesc_hasClause(?methodName, ?c) ∧InvokeClause(?c) ∧InvokeClause_invokedObject(?c, ?o) ∧Object_hasType(?o, ?n) ∧
name(?n, "Thread") ∧InvokeClause_invokedMethod(?c, ?m) ∧
name(?m, "start") ∧NotCallStartInConstructMethodWarning(?w)
 → errorReport(?c, ?w)
```

Fig. 4. The precise rule definition in SWRL

4 Modeling Source Code and Vulnerability Reasoning

After the Java specification and bug patterns are modeled, Java source code should be modeled to detect bugs. Taking the execution times and memory consumption into account, we will not map all the Java elements to OWL individuals. Instead, we only model those elements within our domain of interests. To reduce the number of individuals, program slicing technique is used. The process is described in the Figure 5.

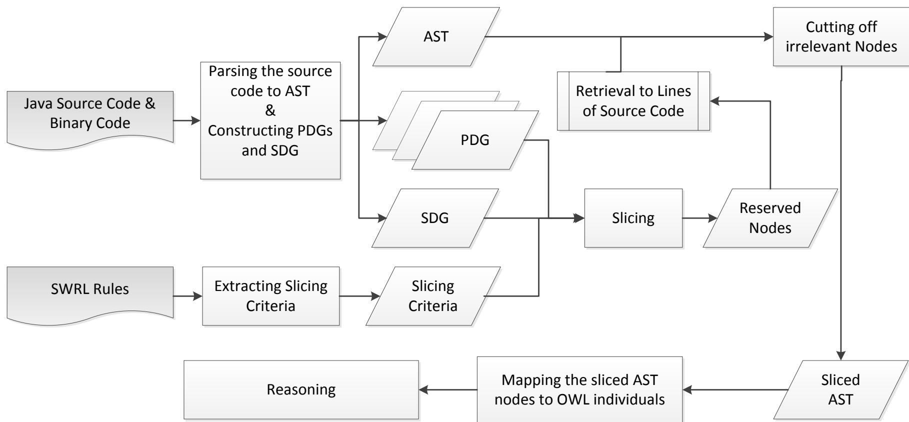

Fig. 5. The process of modeling source code and slicing

The five steps of this phase are described as bellow, and we will discuss them in details in the following sections.

1. Extracting slicing criteria automatically from the SWRL rules.
2. Constructing PDGs for the procedures (methods) and the SDG for the entire system.
3. Once the dependence graphs have been constructed, slicing task is performed over the dependence graphs using the criteria automatically extracted from the SWRL rules.
4. Adopting the slicing results to AST and removing irrelevant nodes, and mapping the rest of AST nodes to OWL individuals.
5. Performing vulnerability reasoning based on the extracted SWRL rules and created OWL individuals from the sliced nodes.

4.1 Slicing over Method and Variable

Program Slicing [2] is a program analysis technique that reduces programs to those statements that are relevant for a particular computation.

This paper uses program slicing algorithms to slice Java source at two levels: method level and variable level. At the method level, it creates call graph based on control flow analysis, the slicing criterion is a list of method name. The slicing result only contains the methods in the criterion list and the methods that call them. At the

variable level, it analyzes control flow and data flow information of each method, and constructs dependence graphs, for example, the procedure dependence graphs (PDGs) and the system dependence graph (SDG) [3]. The slicing criterion at variable level is a list of variables in a method, and the slicing result is all statements which can influence the variable values or can be influenced by the variables.

Figure 6 shows an example of how to slice a class with slicing criteria at the method level. Figure 6(a) is the source code and Figure 6(b) is the slicing result when the slicing criterion is “callMethod” method. Because of the “callMethod” invokes “call” method, the “call” method is reserved, while “notCall” method is cut off.

```
1 public class CallMehtod{
2     public void callMethod() {
3         try {
4             call("");
5             // ...
6         } finally {
7             // ...
8             System.gc(); // VIOLATION
9         }
10    }
11
12    public void call(String op){
13        //Do Something
14    }
15
16    public void notCall(String op){
17        //Do Something
18    }
19 }
```

(a)

```
1 public class CallMehtod{
2     public void callMethod() {
3         try {
4             call("");
5             // ...
6         } finally {
7             // ...
8             System.gc(); // VIOLATION
9         }
10    }
11
12    public void call(String op){
13        //Do Something
14    }
15
16
17
18
19 }
```

(b)

Fig. 6. Static program slicing example, slicing criterion = “callMethod”

The source code in Figure 7(a) is used to calculate $\sum_{i=1}^{n} i$ and $\prod_{i=1}^{n} i$, if the slicing criterion is <12, product> and the strategy is at the variable level, the slicing result will look like the one shown in Figure 7(b). All the statements that influence the variable *product* are reserved. Statements (4), (7), and (10) are not related with variable *product*, so they will be cut off.

```
2     public void calculate(int n){
3         int i = 1;
4         int sum = 0;
5         int product = 1;
6         while(i <= n){
7             sum = sum + i;
8             product = product * i;
9             i = i + 1;
10        }
11        System.out.println(sum);
12        System.out.println(product);
13    }
```

(a)

```
2     public void calculate(int n){
3         int i = 1;
4
5         int product = 1;
6         while(i <= n){
7
8             product = product * i;
9             i = i + 1;
10        }
11
12        System.out.println(product);
13    }
```

(b)

Fig. 7. Static program slicing example, slice criterion = <12, product>

Both the examples in Figure 6 and Figure 7 achieve to reduce the number of Java elements. The Java elements sliced will not be mapped to OWL individuals, thus it reduces the reasoning complexity.

For practical use, both slicing levels need inter-procedural analysis, and slicing over variable is fine-grained and obviously much more complex. We will discuss the variable level slicing in more details in the following sections.

4.2 Extracting Program Slicing Criterion

Each bug or vulnerability pattern can detect specific types of bugs. A vulnerability SWRL rule can be used to identify Java elements. In program slicing terminology, we call these Java elements as a slicing criterion.

Program slice criterion can be automatically extracted from the SWRL rules. One example of the program slice criterion is shown in Figure 4. From the SWRL rule statements name(?n, "Thread") and name(?o, "Start"), we can infer that this slicing criterion has two concerns related to "Thread" and "start". If we are using method level slicing strategy, methods that contains the keyword "Thread" or "start" will be reserved first, and other methods that do not contain the keywords and have no effect on global variables used in reserved methods will be cut off.

Take the typical SQL injection vulnerability as a more detailed example. The primary means of preventing SQL injection are sanitizing and validating untrusted input and parameterizing the query. The code example shown in Figure 8 permits an SQL injection attack because the SQL statement sqlString accepts un-sanitized input arguments.

```
class Login {
  public Connection getConnection() throws SQLException {
    DriverManager.registerDriver(new com.microsoft.sqlserver.jdbc.SQLServerDriver());
    String dbConnection = PropertyManager.getProperty("db.connection");
    // can hold some value like "jdbc:microsoft:sqlserver://<HOST>:1433,<UID>,<PWD>"
    return DriverManager.getConnection(dbConnection);
  }

  String hashPassword(char[] password) {
    // create hash of password
  }

  public void doPrivilegedAction(String username, char[] password) throws SQLException {
    Connection connection = getConnection();
    if (connection == null) {
      // handle error
    }
    String pwd = hashPassword(password);

    String sqlString = "SELECT * FROM db_user WHERE username = '" + username +
                       "' AND password = '" + pwd + "'";
    Statement stmt = connection.createStatement();
    ResultSet rs = stmt.executeQuery(sqlString);

    if (!rs.next()) {
         throw new SecurityException("User name or password incorrect");
    }

    // Authenticated; proceed
  }
}
```

Fig. 8. SQL injection vulnerability example

The SWRL rule for SQL injection is shown in Figure 9. There are 11 variables defined in this rule (*?c, ?s, ?str, ?m, ?p, ?clause, ?executeQuery, ?sql_p, ?asc, ?plus, ?w*), the "*name*" property is directly related to the source code and can be used to extract slicing criterion. From the rule statement *name(?executeQuery, "executeQuery")* and *InvokeClause_invokedMethodParameter(?executeQuery,? sql_p)* we can infer the pattern statement *stmt.executeQuery(sql_string)*. Taking this pattern to the source code in Figure 8, the Java elements to construct slicing criterion can be identified as the statement "*ResultSet rs = stmt.executeQuery(sqlString)*".

Thus if we are taking backward slicing form, the statements that have some effect on the slicing criterion (the *stmt* and *sqlString* variable) will be reserved.

```
SWRL Rule
ClassDesc(?c) ∧ClassDesc(?s) ∧name(?s, "java.sql.Statement") ∧ClassDesc(?str) ∧name(?str, "String") ∧ClassDesc_hasMethod(?c, ?m) ∧
MethodDesc_hasParameter(?m, ?p) ∧MethodDesc_hasClause(?m, ?clause) ∧
InvokeClause_invokedMethod(?clause, ?executeQuery) ∧name(?executeQuery, "executeQuery") ∧
InvokeClause_invokedObject(?clause, ?stmt) ∧Object_hasType(?stmt, ?s) ∧
InvokeClause_invokedMethodParameter(?executeQuery, ?sql_p) ∧
Object_hasType(?sql_p, ?str) ∧AssignClause(?asc) ∧MethodDesc_hasClause(?m, ?asc) ∧
AssignClause_hasLeftExpression(?asc, ?sql_p) ∧ExpressionClause(?exp) ∧AssignClause_hasRightExpression(?asc, ?exp) ∧
Plus(?plus) ∧ExpressionClause_hasOperator(?exp, ?plus) ∧ExpressionClause_hasRight(?exp, ?p) ∧SQL_Injection_Warning(?w)
 → errorReport(?c, ?w)
```

Fig. 9. A SWRL rule for SQL injection detection

4.3 Slicing over the Dependence Graphs

Now we have got the criterion, the next step is to construct the procedure dependence graphs (PDGs) and the system dependence graph (SDG) of the program. This work is done by a slicing module. For now, the PDGs and SDG are constructed based on the work described in paper [4] which is based on the ASM [5] library. ASM is a Java byte-code engineering library. We have started to build our own library to construct dependence graphs from AST which can be built from source code.

For the purpose of reducing memory usage, we use a two-dimensional array matrix to store the procedure dependence graphs. The matrix is N * N size where N is the number of the nodes of the program. Let $Matrix_{i,j}$ be the element in row i and column j in PDG matrix. $Matrix_{i,j} = 0$ means $node_i$ has no dependence relationships with $node_j$. $Matrix_{i,j} = 1$ means $node_i$ is data dependent on $node_j$. $Matrix_{i,j} = 2$ means $node_j$ is control dependent on $node_j$.

SDG is designed as a map data structure in the slicing module. The keys of the SDG map are the call nodes of the entire system, while the values of the SDG map are the PDGs corresponding with the key.

After PDG and SDG have been constructed, the criteria are transformed to the nodes in the dependence graphs, and then the slicing module performs cross-functional slicing task. Only the nodes that are relevant with the criterion node will be reserved. Relevant nodes mean the nodes that are directly or indirectly data dependent or control dependent on the given node together with those nodes that can have effect on the given node.

Take the SQL injection vulnerability in Figure 8 as an example. We have the criterion "*ResultSet rs = stmt.executeQuery(sqlString);*" be automatically extracted based on the SWRL rules. When the slicing module traverses the dependence graphs, the assignment node of *"String sqlString ="* and *"Statement stmt ="* will be detected and reserved because they have effect on the criterion statement. To be more precise, the criterion nodes of *"sqlString*" and *"stmt"* are data dependent on the assignment nodes. Nodes like *"if (!rs.next())"* and *"new SecurityException"* will also be reserved because they are directly or indirectly dependent on the criterion node. Each node in dependence graph has a "line" attribute which indicates the node's line number in source code. Using this attribute we can retrieve the slicing results back to source code and AST. Irrelevant statements such as *"if (connection == null){}"*will be removed from the AST. The sliced AST will be then mapped to OWL individuals.

4.4 Generating OWL Individuals and Vulnerability Reasoning

Once the AST has been sliced, the reserved nodes will be mapped to OWL individuals. We build a parser based on JavaCC to translate the source code into an AST and an OWL generator to visit AST nodes and map them to OWL individuals. JavaCC provides some basic support for the visitor design pattern. For object-oriented programming, the visitor pattern enables the definition of a new operation on an object structure without changing the classes of the objects. In our approach, the "new operation" is to map the nodes to OWL individuals. We defined many visitor classes to visit different nodes and create corresponding OWL individuals.

Since all Java classes begin with a class or interface declaration, a "*JavaTreePars erForJessVisitor*" which implements the *visit(ClassOrInterfaceDeclaration, Object)* method of the *Vistor* interface is constructed as the entrance visitor.

Once the *JavaTreeParserForJessVisitor* object is passed to the accept method of the *CompilationUnit*, which represents the AST of the source code, the visitor will begin to traverse the nodes of AST and create corresponding OWL individuals.

The *JavaElementToJessMapping* class is the key class we have built to help create OWL individuals for source code. It holds a member of *JenaOWLModel* which contains all the Java elements that are defined in the "modeling the Java specification" phase. It also contains a set of create methods to create all the individuals.

For example, there is an OWL class defined as "*ClassDesc*". The *ClassDesc* class is used to represent the concept of "*Class*" in Java program language (Interface is a special kind of class). Once the *JavaTreeParserForJessVisitor* is accepted, the *visit(ClassOrInterfaceDeclaration, Object)* that it implements will be called to create an *OWLIndividual* of the *ClassDesc* class. Now we have created an individual which represents the class that is being analyzed. As the creation process continues, more individuals that represent the methods, statements and variables will be created.

So far, the OWL model for Java specification, SWRL rules for vulnerabilities, such as the rule in Figure 9 Section 4.2, and OWL individuals for Java elements are ready. The remaining task is to load these OWL classes, individuals and SWRL rules into the rule engine and let the rule engine perform the reasoning task. If some of the individuals violate the rules, the reasoning engine will assert an "errorReport" property which is considered as a bug or vulnerability.

5 Experiments and Analyses

In order to facilitate ontology model-based static analysis approach, a prototype is developed named SVD, we have carried out some experiments to validate the approach.

In general, if mapping all Java elements to ontology individuals, there will be a lot of individuals that actually will not be used in the reasoning step. What's more, the unused individuals may result in false positives. Program slicing is a mature traditional static analysis method. It can ensure that code within our domain of interest will be reserved while others cut off. We mark the system which is not combined with program slicing module as SVD1, and the current system combined with program slicing module as SVD2.

Table 3 shows the experimental results about running time of SVD1 and SVD2. Within column of SVD1 (or SVD2), Total Time indicates the time (in HH:MM:SS format) that consumes to detect each target, and Total Bugs are the bug number that the systems find.

Table 3. Comparing bug detecting capabilities between SVD1 and SVD2

Name	Number of Class Files	SVD1		SVD2	
		Total Time	Total Bugs	Total Time	Total Bugs
Soot-2.4.0	1134	21:29	187	12:46	184
Struts-2.1.6	753	11:34	164	08:04	162
DOM4j-1.6.1	179	16:08	98	08:43	96
MySQL Connector 3.2.0	122	10:53	58	6:29	57

In Table 3, system SVD2 uses only half of time that SVD1 does to detect the bugs, although the total bug number is slight less than SVD1. In order to figure out how the program slicing module improves the system performance. We take the DOM4j as the target, and carry out another experiment.

Table 4 shows experimental results about time of every part in SVD1 and SVD2. The system without program slicing module shows that reasoning time is accounting about 68 percent of the total time, loading time is accounting about 21 percent of the total time, and they take almost 90 percent of the total time. But after combined with program slicing, they only take 67 percent of the total time. What's more, the total run time of the system reduces 45.9 percent, and the program slicing only takes 13 percent of the total time. From analyzing the Table 4, it can be seen that combing with program slicing can greatly improve the performance.

Table 4. Comparing performance of SVD1 and SVD2

Time/System	Slicing	Parsing	Loading	Reasoning	Total Time
SVD1	0	1:47	3:23	10:58	16:08
SVD2	01:08	01:38	01:56	04:01	08:43

6 Related Work

Algorithms and techniques for source-code analysis have been changed, sometimes dramatically, for more than thirty years. But the anatomy of source-code analysis has always been the same. David Binkley [6] summed up the three components of source-code analysis as parser, internal representation and the analysis of this representation.

Parsers convert source code, the concrete syntax, into internal representations which are in abstract syntax and are better suited to a particular analysis. Most parsers are compiler-based and process the entire language [8]. There are lighter-weight techniques that handle only part of the language. For example, an island grammar [9] allows portions (islands) of the concrete syntax to be parsed while ignoring the remainder. Our approach also provides a means to control the scale of the language syntax by constructing the ontology models of language specification, although we consider it unnecessary in most cases.

The internal representation abstracts a particular aspect of the program into a form more suitable for automated analysis. Common examples of internal representation include the control flow graph, the abstract syntax tree (AST) and the call graph. Static single-assignment (SSA) form [10] is another popular internal representation. SSA form simplifies and improves the precision of a variety of data-flow analyses [6].

Graph is the most common internal representation. For example, the value dependence graph (VDG) represents control flow as data flow and thus simplifies analysis [11]. Dynamic call graphs [12] [13] and XTA graphs built in support of dynamic reachability-based inter-procedural analysis [12]. Trace Flow Graph (TFG) consists of a collection of CFGs with additional vertices and edges to represent inter-task control flow, and is used to represent concurrent programs [14].

In event-driven systems or distributed programs, finite state automata (FSA) is usually used to represent the analysis, FSA provides an excellent abstraction of program models [15].

For the consideration of interoperability, there are some internal representations that are "external" to the individual tools. For example, srcML (http:// www. sdml.info/index.html), which is a combination of source code (text) and selective AST information (tags) in a single XML document. SrcML is taken to support understanding, analysis, and transformation of large software systems undergoing evolution. In our ontology model-based analysis approach, program models are represented as first AST and then ontology models, ontology models have the advantage same as srcML or XML, since OWL is a W3C standard, program models built in form of ontology models can be understood by all the tools implementing the standard.

The third part of the component in source-code analysis is the actual analysis itself. Analyses can be classified along six dimensions [6]: static versus dynamic, sound versus unsound, safe versus unsafe, flow sensitive versus flow insensitive, context sensitive versus context insensitive, and complexity. A slightly different classification that focuses on object-oriented analysis is discussed by Ryder [16]. There are almost

as many analyses as there are internal representations. Most analysis problems have a spectrum of solutions that represent precision-effort trade-offs. For example, imprecise points-to sets can be computed in near linear time, while the computation of flow- and context- sensitive points-to sets is NP-hard [17]. To discuss the details of each analysis methodology is beyond this paper's topic. In our ontology model-based analysis approach, the analysis task, or more accurately the reasoning task, is delegated to reasoning engines that implement OWL and SWRL standards. This brings up the versatility to apply this approach to different languages.

7 Conclusion and Future Work

This paper proposes program slicing-enhanced ontology model-based static analysis for security vulnerability detection. Programming language specification is modeled as OWL classes and properties, and vulnerability patterns are modeling in rules in SWRL. Java source code is parsed into AST and then mapped into OWL individuals. The huge number of OWL individuals impacts significantly the performance the vulnerability reasoning. To reduce the number of OWL individuals, program slicing technique is introduced to remove the irrelevant OWL individuals regarding to a specific type of vulnerability, thus reducing the reasoning complexity. We developed a prototype tool named SVD to show the validity of the proposed static security vulnerability analysis approach.

Detection capability to a large extent depends on the number of OWL individuals and SWRL rules. To realize a full-fledged SVD tool, we will keep on implementing SWRL rules and OWL classes for more bug patterns, and writing AST visitors to generate the corresponding individuals. We are planning to adopt more other traditional static analysis technologies such as symbolic execution, type inference, and interval analysis, together with the ontology model based approach.

Acknowledgments. The research is partially supported by the National Science Foundation of China (No. 60973001, No. 61100047).

References

[1] Yu, L., Zhou, J., Yi, Y., Li, P., Wang, Q.: Ontology Model-Based Static Analysis on Java Programs. In: Proceedings of the 2008 32nd Annual IEEE International Computer Software and Applications Conference, July 28-August 01, pp. 92–99 (2008)
[2] Weiser, M.: Program slices: formal, psychological, and practical investigations of an automatic program abstraction method. PhD thesis, University of Michigan, Ann Arbor (1979)
[3] Ferrante, J., Ottenstein, K., Warren, J.: The program dependence graph and its use in optimization. ACM Transactions on Programming Languages and Systems 9(3), 319–349 (1987)
[4] Java System Dependence Graph API, http://www4.comp.polyu.edu.hk/~cscllo/teaching/SDGAPI/
[5] ASM, http://asm.ow2.org/

[6] Binkley, D.: Source Code Analysis: A Road Map. In: 2007 Future of Software Engineering (FOSE 2007), pp. 104–119. IEEE Computer Society, Washington, DC, USA (2007), doi:10.1109/FOSE.2007.27
[7] Cordy, J., Dean, T., Malton, A., Schneider, K.: Source transformation in software engineering using the TXL transformation system. Information and Software Technology 44(13) (2002)
[8] Edison Design Group. Compiler front ends (2006)
[9] Moonen, L.: Generating robust parsers using island grammars. In: Working Conference on Reverse Engineering (2001)
[10] Cytron, R., Ferrante, J., Rosen, B., Wegman, M., Zadeck, K.: Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Prog. Lang. Syst. 13(4) (1991)
[11] Weise, D., Crew, R.F., Ernst, M., Steensgaard, B.: Valuedependence graphs: Representation without taxation. In: Conference Record of POPL 1994: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. ACM SIGACT and SIGPLAN, ACM Press (1994)
[12] Qian, F., Hendren, L.: Towards dynamic interprocedural analysis in jvms. In: Proc. of the 3rd Virtual Machine Research and Technology Symposium, San Jose, USA. Usenix (May 2004)
[13] Pheng, S., Verbrugge, C.: Dynamic data structure analysis for Java programs. In: ICPC 2006: Proc. of the 14th IEEE International Conference on Program Comprehension. IEEE Computer Society (2006)
[14] Cobleigh, J., Clarke, L., Osterweil, L.: Flavers: A finite state verification technique for software systems. IBM Systems Journal – Software Testing and Verification 41(1) (2002)
[15] Schmidt, D.: Structure-preserving Binary Relations for Program Abstraction. In: Mogensen, T.Æ., Schmidt, D.A., Sudborough, I.H. (eds.) The Essence of Computation. LNCS, vol. 2566, pp. 245–265. Springer, Heidelberg (2002)
[16] Ryder, B.: Dimensions of Precision in Reference Analysis of Object-oriented Programming Languages. In: Hedin, G. (ed.) CC 2003. LNCS, vol. 2622, pp. 126–137. Springer, Heidelberg (2003)
[17] Landi, W., Ryder, B.G.: Pointer-induced aliasing: A problem classification. In: Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando, FL. ACM Press (January 1991)