A Hybrid Approach to Detecting Security Defects in Programs

Lian Yu†, Jun Zhou, Yue Yi, Jianchu Fan, Qianxiang Wang
School of Software and Microelectronics, Peking University, Beijing, 102600, P.R. China
†Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, P.R. China

Abstract

Static analysis works well at checking defects that clearly map to source code constructs. Model checking can find defects of deadlocks and routing loops that are not easily detected by static analysis, but faces the problem of state explosion. This paper proposes a hybrid approach to detecting security defects in programs. Fuzzy Inference System is used to infer selection among the two detection approaches. A cluster algorithm is developed to divide a large system into several clusters in order to apply model checking. Ontology based static analysis employs logic reasoning to intelligently detect the defects. We also put forwards strategies to improve performance of the static analysis. At last, we perform experiments to evaluate the accuracy and performance of the hybrid approach.

Keywords: security defects, static analysis, model checking, ontology model, fuzzy inference, feature extraction.

1. Introduction

Software security vulnerability is a common problem in existing applications, which tends to bring serious consequences. Roughly half of all security defects are introduced at the source code level [1], and coding errors are a critical problem. It is not reasonable to expect developers to be security experts, and we must arm them with the tools to make their jobs easier. Among many methods, static analysis and model checking have been used to automatically detect potential security problems.

Static techniques explore abstractions of all possible program behaviors, and thus are not limited by the quality of test cases to achieve high quality of programs. Many static analysis tools can be used to uncover security defects. ITS4 [29], FlawFinder [26], and RATS [38] all preprocess and tokenize source files and then match the resulting token stream against a library of vulnerable constructs. FindBugs [25] is a lightweight checker for unearth common errors in Java programs.

Recent research shows that static analysis works well at checking defects that clearly map to source code constructs [2]. Static analysis checker can process a lot of source code with a fast speed compared to model checking. It can also analyze code which is not completed or with compiling errors. The tools look for a fixed set of defect patterns, or rules, in the code. However, if a rule has not been written yet to find a particular problem, the tools will never find that problem. As static analysis problems are undecidable, it is forced to make approximations and that these approximations lead to less-than-perfect output [3][4].

Model checking is capable of handling complex systems, widely used for the verification of all kinds of systems, especially critical systems. The well-used model checking tools include SPIN [41], SMV [42], JPF [33], Bandera [18], and Microsoft SLAM [40].

Model checking can find defects that are not easily detected by static analysis by executing code, such as deadlocks and routing loops that need to analyze the whole execution of the source code. Model checking checks all executing paths while static analysis checks all possible paths, so program segments containing defects found by static analysis may be unreachable. The defects found by model checking are the true ones, while the static analysis may complain about some false positives.

Model checking has the problem of state explosion. The size of a finite state model increases exponentially as the number of system classes grows. When checking large models, the traces (especially thread-related and protocol-related) produced by the checker can be hundreds, or even thousands of steps long. Using the trace log to locate the defects in the source code is extremely time-consuming and tedious, thus it is difficult to interpret the outputs. Table 1 summarizes the comparison of the two approaches from different aspects.

<table>
<thead>
<tr>
<th>Table 1: Comparison of the Two Approaches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aspects</td>
</tr>
<tr>
<td>Time to detect</td>
</tr>
<tr>
<td>Path coverage</td>
</tr>
<tr>
<td>Time per bug</td>
</tr>
<tr>
<td>Bugs found</td>
</tr>
<tr>
<td>Code size</td>
</tr>
</tbody>
</table>

After classifying the categories of security defects, we find some security defects (D\textsubscript{1}) are suitable to be detected by static analysis; some of them (D\textsubscript{2}) are suitable to be detected by model checking, while some of them (D\textsubscript{3}) can be detected by both approaches. For types of defects from

† The research is supported by the National High-Tech Research and Development Plan of China (No. 2006AA01Z175) and by IBM Joint Research Program (No. J5200705002).
$D_s$, if we apply static analysis, we might miss some defects due to the limitation of the approach; if we apply model checking, it will take a long, even endless, time to get the results.

This paper proposes to introduce fuzzy inference system (FIS) to help automatically select a detection approach. Based on the characteristics of source code, FIS decides the detection approach. Chi-square method is used to extract security defect features from defect descriptions and we map them into reserved words or class names in the standard libraries of a program language. To improve the performance of ontology-based static analysis, we introduce parallel computing and ontology model partitioning. Figure 1 shows the whole process of the proposed hybrid approach to detecting security defects in programs; each step will be detailed in following sections.

![Figure 1. Process of the proposed hybrid approach to detecting security defects](image)

1) Collect manually security defects from security research organizations and related books. Extract the features from the descriptions of security defects and map the features to the reserved words or class names of C++/Java programming languages.

2) Cluster the whole system under test according to the system dependence graph extracted from the source code using clustering algorithm; and divide the whole system into several clusters.

3) Build a Mamdani fuzzy inference system [16] for each detection approach, with the reserved words and classes of a programming language as the input variables, the possibility of a detection approach as the output variables. Pick a cluster from step 2, and make a fuzzy inference which attributes the inputs to a proper approach to detecting security defects.

4) Extract the finite state model from target source code and perform model checking on each cluster.

5) Perform the static analysis based on the logic reasoning. Source code is parsed into an AST, and C++/Java AST Parser maps it into individuals in OWL (Web Ontology Language) [36] model; Reasoning Engine (Jess [31] Engine) takes security rules written in SWRL (Semantic Web Rule Language) [43], programming language ontology model and the individuals created by the tree parser as inputs to perform defect reasoning.

6) Iterate these steps. If a cluster is large and takes a long time to run, go back to step 2 to divide the cluster into smaller clusters. If the current cluster is not the last cluster, pick another cluster and go back to step 3 to make a fuzzy inference.

7) Generate the final report on security defect detection.

The rest of the paper is organized as follows. Section 2 describes classification, feature extraction for security defects, and algorithms to cluster programs under detection. Section 3 explains the mechanism of fuzzy inference based selection of detection approach. Section 4 demonstrates security detection using static analysis. Section 5 applies model checking to detect security defects. Section 6 puts forward strategies to improve the performance of the static analysis. Section 7 carries out experiments to evaluate the proposed approach. Section 8 gives a survey of related work. Section 9 concludes the paper and scratches our future work.

2. Classification, Extraction, and Clustering

2.1. Classifying Security Defects

At first, we collect the descriptions of security defects from popular websites, related books [4][5] and papers.
CWE (Common Weakness Enumeration) [22] is a very famous and authoritative Website related to software weakness. It contains various types of defect domains. Because we focus on the code level defects, we only abstract source code related security defects. CVE (Common Vulnerability Enumeration) [21] is another Website which provides a dictionary of publicly known information security defects. Totally we obtain about 110 security defects, and Table 2 shows two examples of defects associated with security defect descriptions.

<table>
<thead>
<tr>
<th>Security Defects</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unchecked Error Condition</td>
<td>Ignoring exceptions and other error conditions may allow an attacker to induce unexpected behavior unnoticed.</td>
</tr>
<tr>
<td>Use of Insufficiently Random Values</td>
<td>The software may use insufficiently random numbers or values in a security context that depends on unpredictable numbers.</td>
</tr>
</tbody>
</table>

There exist many different classifications for security defects, such as 19 sins [14], OWASP Top 10 [35], and 7PK [15]. We employ the 7PK (Seven Pernicious Kingdoms) which divides all the software security defects into eight categories (Note: seven plus one [7]): input validation and representation, API abuse, security features, time and state, errors, code quality, encapsulation, and environment. According to this classification and the description of the defects we manually tag each defect with a category label and associate each defect with a proper detection approach, model checking or static analysis as shown in Figure 2. When this step is done, the classifying result is used to extract features from these security defects.

2.2. Security Defect Feature Extraction

The feature extraction of security defects helps us to recognize whether a kind of features is suitable to be detected by static analysis or by model checking. There are several algorithms to extract common features in current text classification domain: Document Frequency (DF), Information Gain (IG), Mutual Information (MI), and Chi-square Statistic [8]. These algorithms aim at selecting terms/features which are most comprehensive of text contents. Experiments show that Chi-square is more effective at removing aggressive terms (whose chi-score is lower than predetermined threshold) without losing categorization accuracy [8]. The terms in this paper are words in the description of the security defects, and the category is the detection approaches that contain the eight categories described in the previous subsection. The Chi-square statistic measures the relationship between term $t$ and category $c$, as shown in the following formula:

$$
\chi^2(t, c) = \frac{N \cdot (AD - CB)^2}{(A + C) \cdot (B + D) \cdot (A + B) \cdot (C + D)}
$$

Where $A$ is the number of times that $t$ and $c$ co-occur; $B$ is the number of times that $t$ occurs without $c$; $C$ is the number of times that $c$ occurs without $t$; $D$ is the number of times that neither $c$ nor $t$ occurs, and $N$ is the total number of documents. We can calculate the average score and maximum score by combining the category scores of each term. The maximum score is used to determine which category a feature belongs to, and the average score can distinguish whether the word is an invalid feature.

$$
\chi^2_{avg}(t) = \frac{1}{m} \sum_{i=1}^{m} P_t(c_i) \chi^2(t, c_i)
$$

$$
\chi^2_{max}(t) = \max_{i=1}^{m} \chi^2(t, c_i)
$$

As shown in Figure 2, each detection approach can handle some defects from the 7PK categories and each 7PK category contains defects, some of which can be detected by static analysis or by model checking or by both. For each detection approach, we apply chi-square testing to extract security defect features from defect descriptions associated with the 7PK categories as shown in Table 3. These features will be used in fuzzy inference in the next section.

<table>
<thead>
<tr>
<th>Detection Approach</th>
<th>Features/Terms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model Checking</td>
<td>buffer overflow username input character char string pointer stream exception network locked integer asynchronous threads read process</td>
</tr>
<tr>
<td>Static Analysis</td>
<td>path random form Struts private public filenames print!Action exceptions inject returns J2EE array dirname session EJB connection cloneable</td>
</tr>
</tbody>
</table>

2.3. Clustering Programs under Detection

Clustering serves two purposes. By building a system dependence graph (SDG) from source code and cutting the SDG into smaller parts, the size of the system model can be reduced to a suitable one in order to run model checking. Static analysis can also take advantage of SDG to increase its accuracy. The clustering algorithm contains the following four steps.

1) Build a dependence matrix. Its smallest unit is a class.

Set the number of classes as the size of the matrix. The entries in the matrix are the count of the dependence relationship among the classes. The count is calculated as follows, where $a1$, $a2$, and $a3$ are constants, $inv$ is the invocation count, $impl$ is the number of interface implementations, and $ext$ is the number of extending classes.
count = a1 × inv + a2 × impl + a3 × ext

2) Randomly select a class, and compute the cost $d_{value}(\text{class}_i, \text{class}_j)$ associated with each cluster $\text{cluster}_k$ as shown in the following formula (E-1) where $pow_{dep}$ and $pow_{cc}$ are constants. Sort $d_{value}$ and put the unit into a cluster with largest $d_{value}$.

$$
\sum_{j \in \text{class}_k} \frac{(\text{Matrix}(t, j) + \text{Matrix}(j, i))^{pow_{dep} \cdot \text{cluster}_k \cdot size(k)^{pow_{cc}}}}{\text{Matrix}(t, j)\text{Matrix}(j, i)}
$$

(E-1)

3) Calculate the matrix cost, $Cost(\text{class}_i)$, which is used to describe current clustering result. The smaller the value is, the better the clustering result is. When the matrix is changed, its cost should be recalculated as shown in the following formula (E-2). Compare the cost with the previous one. If the new cost is smaller than the old one, accept the result and update the matrix.

$$
\begin{cases}
\sum_{j \in \text{class}_k} (\text{Matrix}(i, j) + \text{Matrix}(j, i))^{pow_{dep} \cdot \text{cluster}_k \cdot size(k)^{pow_{cc}}} \\
\text{if } i, j \text{ are in the same cluster}
\end{cases}
$$

$$
\begin{cases}
\sum_{j \in \text{class}_k} (\text{Matrix}(i, j) + \text{Matrix}(j, i))^{pow_{dep} \cdot \text{cluster}_k \cdot size(k)^{pow_{cc}}} \\
\text{if } i, j \text{ are not in the same cluster}
\end{cases}
$$

(E-2)

4) Iterate the steps 2 and 3, and finally get a stable matrix, which consists of several clusters and will be used by model checking to detect security defects.

3. Fuzzy Inference-based Selection

This section employs fuzzy inference techniques to determine dynamically which detection approach should be used to identify security defects for each cluster which is produced in the previous section.

3.1. Mapping Features

Because those extracted features cannot be used directly by the detection approaches, the step of mapping the extracted features to the reserved words and classes of programming language is needed. Table 4 shows these mapping results in Java language. MC indicates model checking and SA static analysis. Currently, the mapping is manually performed, and our future work will address developing automatic mapping algorithms and proving the completeness and correctness of the algorithms.

**Table 4: Mapping Features to Java Characteristics**

<table>
<thead>
<tr>
<th>MC-related Features</th>
<th>Reserved Words or Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>character char string</td>
<td>String/Char/StringBuffer/StringBuilder</td>
</tr>
<tr>
<td>pointer</td>
<td>(All function Invoking)</td>
</tr>
<tr>
<td>buffer overflow</td>
<td>N/A</td>
</tr>
<tr>
<td>stream read</td>
<td>All classes in the package java.io</td>
</tr>
<tr>
<td>exception</td>
<td>All Exception</td>
</tr>
<tr>
<td>network</td>
<td>All Classes in the package java.net</td>
</tr>
<tr>
<td>locked asynchronous</td>
<td>synchronized/All Classes in the package java.util.concurrent</td>
</tr>
<tr>
<td>integer</td>
<td>Integer, int</td>
</tr>
<tr>
<td>threads process</td>
<td>Thread</td>
</tr>
<tr>
<td>SA-related Features</td>
<td>Reserved words or Classes</td>
</tr>
</tbody>
</table>

**Random**

Form Struts Action

ActionForm, and the classes related with Struts framework

private public

private public protected

path filenames

dirname java.io.File

exceptions All Exception

returns return

J2EE EJB

All classes related with EJB framework

connection All Classes in the package java.net

cloneable java.lang.Object

inject JDBC and XML operation classes

3.2. Building a Fuzzy inference System

We create two FISs for model checking and static analysis, respectively. Building an FIS can be divided into 5 steps: 1) Fuzzify inputs 2) Apply fuzzy operator 3) Apply implication method 4) Aggregate all outputs 5) Defuzzify. Mamdani fuzzy inference method is the most commonly and useful fuzzy methodology [17]; and we use the Mamdani method for each FIS.

**Step 1 Fuzzify inputs:** Use Gauss member function (E-3) to fuzzify each input parameter based on Table 4. Each parameter contains three member functions named low, median and high, which show the possibility (from 0 to 1) of reserved words or classes contained in a program. Each FIS contains nine input parameters shown in the “Reserved Words or Classes” column in Table 4 and one output parameter. Figure 3 shows the model checking FIS as an example. The input variables are “string”, “pointer”, “buffer”, “stream”, “exception”, “network”, “lock”, “integer”, and “thread”.

$$
f(x) = ae^{-(x-b)^2/c^2}
$$

(E-3)

**Figure 3:** Model checking Fuzzy Inference System

**Step 2 Apply fuzzy operators:** To apply fuzzy operator, we use the min function for AND method and the max function for OR method.

**Step 3 Apply implication method:** The min function is used as an implication method. We choose the min value from all the input parameters.

**Step 4 Aggregate all outputs:** Because decisions are based on the testing of all rules in an FIS, we have to combine the rules in a manner in order to make a decision. This paper selects the max function as the aggregation function.

**Step 5 Defuzzify:** The defuzzification applies the most
popular used centroid approach, which returns the center of area under the output curve.

The fuzzy rules are fabricated manually from the security defects descriptions as shown in Table 2. There are eight rules for model checking FIS and eleven rules for static analysis FIS. For example, the following rule says that it is more suitable to select model checking approach if the thread class has a high appearance possibility, the synchronized keyword has a low appearance possibility and the stream class has a high appearance possibility.

\[
\text{If (thread is high) and (synchronized is low) and (Stream is high) then (model checking is high)}
\]

For each cluster, we obtain an output from the model checking FIS and/or static analysis FIS. It ranges from 0 to 1. If the output value from model checking FIS is high, we conclude it should be analyzed by model checking method, and if the value for static analysis FIS is high, it should be analyzed by static analysis. If both are high, we check the class with both methods.

4. **Detecting Security Defects with Static Analysis**

If static analysis is selected by FISs to find the security defects in source code and byte code, we use ontology based approach to accomplish static analysis. The process is similar to our previous paper [9], but we add more SWRL rules into the rule base to detect vulnerability related bugs.

4.1. **Ontology-based Detection Process**

We develop a parser to translate source code into an Abstract Syntax Tree (AST), and a module to map the AST instance to the individuals of Java ontology model, which is constructed according to the programming language specifications. SWRL Bridge takes the individuals, defect rules and the programming language ontology model as inputs into the reasoning engine to infer defects. Finally a report is generated.

4.2. **Building Security Defect Rules**

In Section 2.1, we have explored the security databases of CVE and the CWE, and abstracted the security defects. This section transforms these defects into the rules using SWRL and put these security defect rules into a database for dynamically loading and reasoning. We collect about 60 security defects that are suitable to be detected by static analysis. The following shows two transformation examples of security defects into SWRL rules.

**Rule for Use of Dynamic Class Loading** is:

\[
\text{InvokeClause(?c)} \land \text{invokeClause_invokedMethod(?c, ?m)} \land \\
\text{invokeClause_invokedMethodParameter(?c, ?ob)} \land \\
\text{name(?m, "forName")} \land \text{name(?t, "java.lang.String")} \land \\
\text{Object_hasType(?ob, ?t)}
\]

**Rule for Use of Insufficiently Security Random Values** is:

\[
\text{UseDynamicClassLoadingWarning(?warning)} \rightarrow \text{errorReport(?c, ?warning)}
\]

\[
\text{Rule for “Use of Insufficiently Security Random Values”}
\]

\[
\text{Object(?c) \land Name(?m,"java.lang.Random") \land} \\
\text{UseOfInsufficientlyRandomWarning(?warning)} \rightarrow \text{errorReport(?c, ?warning)}
\]

The proposed rules are of the form of an implication between an antecedent (body) and consequent (head). The meaning is read as: whenever the conditions specified in the antecedent hold, then the conditions specified in the consequent must also hold. The detailed description on how to transform can be found in the paper [9].

5. **Detecting Security Defects using Model Checking**

Once model checking is selected by FISs to find the security defects in programs, we utilize the integrated model checking approach to fulfill the goals.

5.1. **Detection Process using Model Checking**

The integrated model checking follows the four steps:
1) Cluster the classes: The program model can be cut into smaller pieces using the clustering algorithm described in Section 2.3.
2) Generate drive and stub classes: Automatically create an enter class and stub classes for each cluster as described in section 5.2.
3) Abstract the model: Extract the finite state model from the programs under detection using proper ways.
4) Perform the model checking: Select a model checking tool to find the defects in the program model and report the results.

5.2. **Automatically Generating Drivers and Stubs**

Although we have already clustered the classes and got smaller pieces, we cannot perform the model checking without an entry point. Each cluster needs main functions to start with the source code, and also needs stubs to simulate the action of the classes that are not contained in the current cluster.

In order to emulate the outer actions to a cluster, we generate driver classes for each public method of a class in this cluster, which is invoked by the classes in other clusters. The responsibility of each enter class is to create a proper instance, and invoke the target public function. Because current model checking tools do not have the ability to create the stubs, we also generate default actions as stubs for each class, which are contained in other clusters and are invoked by the classes in current cluster.

5.3. **Applying Model Checking**

Compared to Bandera, Java Pathfinder (JPF) is of high performance according to our experiments. We employ
JPF to perform model checking in this step. JFP is open source software, an explicit state model checking tool for Java code developed by NASA Ames Research Center. It simulates the JVM, automatically abstracts the finite state model from the code, and then checks whether there is some vulnerability in the model.

6. Strategies to Improve Performance

We performed several experiments on our ontology-based static analysis tool and found its low performance issue [9]. The proportion of time consumed is shown in Figure 4, where up to 70% of time is used for reasoning, 20% is for loading, and 10% is used to parse Java source code and other processing.

![Figure 4. Time Allocations among three parts](image)

Let $T$ be the total time, $R$ the reasoning time, $L$ the loading time, $P$ the parsing time, and $O$ the other time. The proportion of time consumed can be calculated in the formula: $T = R + L + P + O$. To improve its performance, we design strategies to cut down reasoning time, loading time and parsing time, respectively.

6.1. Parallel Computing

The previous version of the ontology-base static analysis tool is single-threaded; processing source code or bytecode files one by one as follows in Figure 5:

![Figure 5. Single-Thread Architecture](image)

Workstations or PCs nowadays generally have more than one CPU. We propose to change the single-threaded architecture to a multi-threading architecture to take the advantage of multi-CPU. We use a thread pool to limit the maximum amount of the running threads. The thread pool maintains a queue, which contains the IDs of running threads that parse concurrently multiple source code or byte-code and produce a set of ASTs as shown in Figure 6. The ASTs are sorted using a sorting algorithm that will be introduced in Section 6.2, and then the parallel computing continues.

The size of the queue is the amount of the maximum running threads. The queue is checked several times per seconds, and if there is a finished thread, it starts a new thread and puts it into the queue. In this way, we maintain a fixed amount of running threads, so that not only can we get the good performance, but also do not take up many system resources.

![Figure 6. Multi-Threaded Architecture](image)

6.2. Sorting ASTs according to their Similarity

If a reasoning engine processes ASTs in an order such that rules the ASTs need to check against are almost same, we will reduce the time for a reasoning engine to swap in and out those rules, therefore improving the reasoning performance. To reach this goal, we sort the ASTs before loading them into reasoning engine as follows:

1) Get detection rule list for every source code file, and create an AST – rule relationship matrix as shown in Figure 7, where, $X$ axis represents AST IDs, $Y$ axis represents detection rule IDs. The entry with 1 means that the corresponding AST should be reasoned against the corresponding detection rule.

![Figure 7. AST-Detection Rule Relationship Matrix](image)

2) Create an AST association matrix as shown in Figure 8, where $X$ and $Y$ axes both represent AST, and its entry is the degree of association of the corresponding two ASTs in terms of detection rules that share. Let $B(i, j)$ be the intersection of Rule $i$ and Rule $j$, and $C(i, j)$ the union of Rule $i$ and Rule $j$, the entry value is calculated by formula: $AA[i][j] = B(i,j) + C(i,j)$.

![Figure 8. AST matrix of association](image)

3) Sort ASTs using the following procedure:
   a) Find two ASTs with the maximum degree of association, and add them to an array;
   b) Let $H1$ be the first element in the array, $H2$ be the...
second element. For an AST that has not been added into the array, let \( D1 \) be the degree of association for \( H1 \) and the AST, and \( D2 \) be the degree of association for \( H2 \) and the AST, calculate a value using the formula: \( D1 \times \alpha + D2 \times \beta \), \( \alpha + \beta = 1 \). By experiments, we find that when \( \alpha \) equals to 0.6 and \( \beta \) equals to 0.4, the performance gets the best. Find the maximum value and the corresponding AST and insert the AST to the head of the array.

c) Let \( T1 \) be the last element in the array, \( T2 \) be the last but one element in the array. For an AST that has not been added into array, let \( D1 \) be value of degree of association for \( T1 \) and the AST, and \( D2 \) be value of degree of association for \( T2 \) and the AST, calculate a value using the above formula, find the maximum value and the corresponding AST and insert this AST to the end of array.

d) Repeat step b and c until all ASTs have been added into the array, which is the sorted AST list. The final order of sorted ASTs is shown in Figure 9.

![Figure 9.Sorted AST Results](image)

It is observed in Figure 9 that after the sorting, two adjacent ASTs have the maximum number of shared rules. This implies that when the reasoning engine switches from processing one AST to another, it reduces to the extreme the number of rule swapping. Therefore, we reach the goal to improve reasoning performance.

6.3. Selective AST Extraction

In the ontology-based static analysis, the loading process consists of the activities: 1) instantiate an OWL model with the Java specification OWL model and the SWRL rules of defect detection; 2) parse Java source code to construct an AST; and 3) extract OWL individuals and OWL properties from the AST to exemplify the OWL model. In the reasoning process, SWRL Bridge takes this OWL model as an input, transforms it from OWL syntax to Jess syntax, and passes the results on to Jess reasoning engine to do logic reasoning, and at last the engine returns asserted results back to SWRL Bridge.

Before reasoning we examine the security detection rules and only extract the relevant or necessary OWL individuals and OWL properties to instantiate the OWL model. Instead of loading all security detection rules into the reasoning engine at a time as we did before in our previous paper [9], we change the design to load one rule at a time, and the overall process is described as follows:

1) Instantiate an OWL model only with the Java specification.
2) Extract all detection rules from another OWL file with SWRL rules, which are created by Protégé in the phase of Bug Pattern modeling [9].
3) Load one rule and create the obligatory OWL individuals and properties from an AST for the rule.
4) Perform SWRL-to-Jess syntax transformation using SWRL Bridge, and run Jess reasoning engine.
5) Clear up the loaded rule, OWL individuals and properties, and prepare for the next rule.

With this design, we performed a simple evaluation on a Java source code with 1,800 lines of code, reasoning time is reduced from 63,141ms to 16,485ms, but loading time increases from 20,406ms to 176,144ms. It is observed that the new reasoning time is only about one fourth of the old time. We achieve the goal of shortening the reasoning time. However, the loading time increases.

After analyzing, we conclude that the increased loading time is mostly spent on repeatedly creating the same OWL individuals and OWL properties, since every time a new rule is loaded, the necessary OWL individuals and OWL properties associated with the rule are extracted and created. With the loop, when different rules are loaded, some same OWL individuals and OWL properties are created over and over again; a lot of time is spent on the operation. To solve this problem, we propose a solution: after the previous rule is reasoned and before the next rule is loaded, analyze the similarity between these two rules and find out which OWL individuals and OWL properties can be reused to avoid the repeated creation.

With this design, we modify the implementation, and carry out an experiment on the same Java source code, reasoning time is reduced from 63,141ms to 16,485ms, and loading time is reduced from 20,406ms to 10,144ms.

6.4. Selectively Rules Loading

We can analyze the characteristics of the target source code, and decide the necessary rules to reason against the target, in this way we can avoid loading and reasoning of unnecessary rules. For example, consider the equals method, when we check a Java source code that does not invoke equals, it is unnecessary to load and run rules related to equals method. We carried out an experiment to verify this assumption. The experiment target is a package org.apache.struts2.components of Struts project, containing 54 Java source files. The total defined rules are 62, therefore, the times to execute the rules are equal to 54*62=3,348. However, we manually analyze these Java source files, and find that the actual required times are only 328. It is obvious that selective loading of rules is very necessary. Based on this consideration, we redesign the overall process as follows:

1) Categorize all defined rules to several groups. Rules in
one group may be related to the same method invocation, e.g., equals, hashCode, or related to the same statement, e.g. if, switch, or describing inheritance errors and etc. Examples of rules categorization are shown in Table 5. Record the categorization results in an XML file with one keyword referring to several rules.

2) Analyze each Java source code and extract characteristics from the corresponding AST of the source code. AST is a structured object and easier to manipulate than source code does. As a result, we obtain a list of keywords describing characteristics of the target.

3) Match the keywords of source code with the rule categories.

4) Load and run the matched rules to check on the source code.

Table 5: Categorization of Rules

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Rules</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>hashCode</td>
<td>DEF_HASHCODE_USE_OBJECT_EQUALS</td>
<td>hashCode invocation</td>
</tr>
<tr>
<td></td>
<td>OverrideEqualsOrHashCode</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MethodErrorChecker2</td>
<td></td>
</tr>
<tr>
<td>toString</td>
<td>ArrayToStringChecker</td>
<td>toString invocation</td>
</tr>
<tr>
<td></td>
<td>ToStringChecker</td>
<td></td>
</tr>
</tbody>
</table>

7. Evaluation of the Proposed Approach

Based on the research work described above, we develop a tool to find the security defects in programs. The tool is an eclipse plug-in, and runs on a Dell PC with a P8600 dual CPU, 3G memory and 160G hard disk. We also install onto the PC JDK1.4, Eclipse 3.2, and MySQL 5.1. In the experiments, we demonstrate program examples under detection only in Java language.

7.1. Detection Accuracy

The first experiment is to evaluate the detection accuracy of the proposed hybrid approach. We develop a P2P client based on Bit Torrent protocol [19] as detection target that contains 43 classes and 3,129 lines of code. We inject about 74 security vulnerabilities into the target, which fall into seven types of security defects as shown in Table 6: 1) Singleton Pattern in a non-thread safe; 2) Inner class containing sensitive data; 3) Passing Mutable objects to un-trusted method; 4) Public static field not marked final; 5) Null pointer related; 6) Array index overflow; and 7) Dead lock. The target is checked using three approaches: model checking only, static analysis only, and the proposed hybrid approach.

It is observed in the Table 6 that the accuracy of the hybrid approach is 98.6%, higher than the pure model checking with 36.5% and pure static analysis with 71.6%. Note that because there are some overlaps in null point related defects, the total percentage of model checking and static analysis is higher than 100%. The time cost is setting between, higher than pure static analysis, and lower than pure model checking. It is expected. We also notice that our tool cannot detect all the null point related security defects. The reason is the driver classes cannot be generated because of some parameters to invoke the cluster is constructed by Factory Design Pattern that hides the information about the class created, and our current tool can not properly generate the driver class. We will continue the research in the future work.

Table 6: Comparison of Detection Accuracy

<table>
<thead>
<tr>
<th>Vulnerability Type</th>
<th>Injected Defects</th>
<th>Model checking</th>
<th>Static Analysis</th>
<th>Hybrid Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type 1</td>
<td>7</td>
<td>N/A</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>Type 2</td>
<td>3</td>
<td>N/A</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Type 3</td>
<td>34</td>
<td>N/A</td>
<td>34</td>
<td>34</td>
</tr>
<tr>
<td>Type 4</td>
<td>3</td>
<td>N/A</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Type 5</td>
<td>18</td>
<td>18</td>
<td>6</td>
<td>17</td>
</tr>
<tr>
<td>Type 6</td>
<td>5</td>
<td>5</td>
<td>N/A</td>
<td>5</td>
</tr>
<tr>
<td>Type 7</td>
<td>4</td>
<td>4</td>
<td>N/A</td>
<td>4</td>
</tr>
<tr>
<td>Total Detected</td>
<td>74</td>
<td>27</td>
<td>53</td>
<td>73</td>
</tr>
<tr>
<td>Time(s)</td>
<td>2.137</td>
<td>224</td>
<td>1,792</td>
<td></td>
</tr>
<tr>
<td>Accuracy</td>
<td>36.5%</td>
<td>71.6%</td>
<td>98.6%</td>
<td></td>
</tr>
</tbody>
</table>

7.2. Time Cost

The second experiment is to observe the trend of time consumes along with increasing the size of source code. The numbers of classes in the projects under detection are 8, 14, 30, 62, 84, 195 and 274, respectively. And their lines of code increase from 1,402 to 41,656. We use three approaches to perform the experiments as shown in Figure 10: 1) static analysis only; 2) model checking only; and 3) fuzzy inference based hybrid approach.

Figure 10. Time consumed when increasing the size

It is observed from Figure 10 that the static analysis approach consumes the lowest time compared with the other two, while the model checking approach without clustering runs out of memory when processing the program with 84 classes and cannot produce results any more. On the other hand, the proposed hybrid approach can gracefully handle the situations due to the introduction of clustering and fuzzy inference based selection among static analysis and model checking. We apply the hybrid approach to other large sizes of programs and find out it can handle up to 400 classes of programs.
7.3. Performance of Static Analysis

This experiment examines the performance improvement of the ontology based static analysis. The projects under detection are dom4j-1.6.1, Struts-2.1.6 and PMD-4.2.4. We compare the detection times before and after applying the strategies described in Section 6. On average, about half the time is reduced as shown in Table 7, in which code size refers to the number of source files.

Table 7: Performance Improvement

<table>
<thead>
<tr>
<th>Project</th>
<th>Code Size</th>
<th>Before (min)</th>
<th>After (min)</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>dom4j-1.6.1</td>
<td>163</td>
<td>8</td>
<td>3.5</td>
<td>56.3%</td>
</tr>
<tr>
<td>Struts-2.1.6</td>
<td>641</td>
<td>28</td>
<td>15</td>
<td>46.4%</td>
</tr>
<tr>
<td>PMD-4.2.4</td>
<td>556</td>
<td>17</td>
<td>9</td>
<td>47.1%</td>
</tr>
</tbody>
</table>

Compared with the results shown in Figure 4, the time allocation of the three parts is more balanced after applying the strategies as shown in Figure 11: reasoning time is 45%, loading time 28%, and parsing time and other 27%.

Figure 11. Time Proportion of Static Analysis

7.4. Time Distribution and Comparison

This experiment reports the time consumed by four computations in the hybrid approach: clustering, fuzzy inference, static analysis and model checking. We choose three projects to examine: dom4j-1.6.1(163 classes), Java Path Finder r1580 (329 classes), and Struts 1.2.7(only check “src\share” folder that contains 272 classes). Table 8 shows the time distributions among the four computations.

Table 8: Time Distribution of the Hybrid Approach

<table>
<thead>
<tr>
<th>Projects</th>
<th>Clustering</th>
<th>Fuzzy inference</th>
<th>Static analysis</th>
<th>Model checking</th>
</tr>
</thead>
<tbody>
<tr>
<td>JPF</td>
<td>26%</td>
<td>1%</td>
<td>13%</td>
<td>60%</td>
</tr>
<tr>
<td>Dom4j</td>
<td>15%</td>
<td>1%</td>
<td>16%</td>
<td>67%</td>
</tr>
<tr>
<td>Struts</td>
<td>28%</td>
<td>1%</td>
<td>13%</td>
<td>59%</td>
</tr>
<tr>
<td>Average</td>
<td>23%</td>
<td>1%</td>
<td>14%</td>
<td>62%</td>
</tr>
</tbody>
</table>

It is observed that more than half the time is spent on model checking, which takes on average 62% of total time. Clustering aims at reducing the complexity of model checking, and burns up 23% of total time. Table 9 shows the defects spotted by the two detection approaches dynamically selected by FISs. The experiment explains that static analysis discovers more defects using less time, and model checking identifies relatively less defects. Furthermore they detect different types of defects and cannot substitute with each other.

Table 9: Detection Comparison

<table>
<thead>
<tr>
<th></th>
<th>JPF</th>
<th>Dom4j</th>
<th>Struts</th>
</tr>
</thead>
<tbody>
<tr>
<td>validation</td>
<td>12</td>
<td>3</td>
<td>23</td>
</tr>
<tr>
<td>representation</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>API abuse</td>
<td>7</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>security features</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>time and state</td>
<td>9</td>
<td>19</td>
<td>2</td>
</tr>
<tr>
<td>errors</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Code quality</td>
<td>0</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>encapsulation</td>
<td>8</td>
<td>0</td>
<td>19</td>
</tr>
<tr>
<td>environment</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>41</td>
<td>27</td>
<td>36</td>
</tr>
</tbody>
</table>
|                | 68   | 39    | 73     | 7

8. Related Work

There are many organizations that provide security defect lists, including CVE, CWE, CAPEC [20], and SCAP [39]. The CAPEC (Common Attack Pattern Enumeration and Classification) is to provide a publicly available catalog of attack patterns along with a comprehensive schema and classification taxonomy. The SCAP (Security Content Automation Protocol) is a method using specific standards to enable automated vulnerability management, measurement, and policy compliance evaluation.

Tools for hunting security vulnerabilities include ITS4 [29], Lint [34] and IBM Rational AppScan [27]. Cigital developed ITS4 to help automate source code review for security. ITS4 statically scans C and C++ source code looking for function calls that are potentially dangerous. For some calls, ITS4 tries to perform some code analysis to determine how risky the call is. In each case, ITS4 provides a problem report, including a short description of the potential problem and suggestions on how to fix the code. It is a command-line tool that works across Unix and Windows platforms.

Lint flags suspicious and non-portable constructs (likely to be defects) in C language source code. It is used to check C code for errors that may cause a compilation failure or unexpected results at runtime. The Lint program issues every error and warning message produced by the C compiler. Many messages issued by Lint can assist in improving program’s effectiveness, including reducing its size and required memory.

IBM Rational AppScan is based on model checking technology. It knows security in-depth and can exploit security defects to prove the impact. It is an eclipse plug-in for Rational Quality Manager and allows QA teams to manage Security Testing just like they manage Quality and Performance testing.

Model checking method emerges as a practical tool for automated debugging of embedded software. There are some survey [10] and some sample model checkers [11], and also some applications to analyze software Java
PathFinder [30], Bandera [18], SLAM [40].

Our method combines static analysis and model checking technologies together to take the advantages of the two approaches. By introducing strategies to improve the performance of ontology based static analysis, developing clustering algorithm to reduce the size of target programs, fuzzy inference guided hybrid approach detect more defects within limited time frame.

9. Conclusion and Future Work

This paper proposes a hybrid approach to automatically detect security defects in programs. We employ the chi-square method to extract the features from the security defect descriptions and map the features into the programming language characteristics. Fuzzy inference is used to intelligently select model checking or static analysis according to the features of programs under detection. We develop a cluster algorithm to divide a whole system into small pieces in order to apply model checking approach when it runs out of computing resources. We use an ontology rule language, SWRL, to build security defect-detection rule base. Several strategies are proposed to improve the performance of the ontology based static analysis. Finally we develop a prototype tool and carry out the experiments to evaluate the proposed hybrid approach. The result shows that our tool can achieve the highest detection accuracy in terms of injected defects compared with the two individual approaches, and significantly cut down the reasoning and loading time when performing static analysis. In addition, the hybrid approach can discover more types of security defects than any of two individual approaches. Although clustering helps reduce the size the detection targets, model checking related computing is still taking up to 80% of the total time. Improving the performance of model checking is our future work.

References

[22] CWE. http://cwe.mitre.org/
[27] IBM Rational AppScan
[33] JPF. http://jpf.sourceforge.net/
[36] OWL. http://www.w3.org/TR/owl-features/
[41] SPIN. http://spinroot.com/
[43] SWRL: http://www.w3.org/TR/swrl/