Frontiers of Information Technology & Electronic Engineering
www.zju.edu.cn/jzus; engineering.cae.cn; www.springerlink.com
ISSN 2095-9184 (print); ISSN 2095-9230 (online)
E-mail: jzus@zju.edu.cn

Efficient vulnerability detection based on an optimized rule-checking static analysis technique*

Deng CHEN†‡1, Yan-duo ZHANG1, Wei WEI2, Shi-xun WANG3,
Ru-bing HUANG4, Xiao-lin LI1, Bin-bin QU5, Sheng JIANG5

(1Hubei Provincial Key Laboratory of Intelligent Robot, Wuhan Institute of Technology, Wuhan 430205, China)
(2Industrial Robot Engineering Center, Wuhan Institute of Technology, Wuhan 430205, China)
(3School of Computer and Information Engineering, Henan Normal University, Xinxiang 453007, China)
(4School of Computer Science and Telecommunication Engineering, Jiangsu University, Zhenjiang 212013, China)
(5School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China)
†E-mail: chendeng8899@hust.edu.cn

Received Nov. 1, 2015; Revision accepted Mar. 20, 2016; Crosschecked Feb. 28, 2017

Abstract: Static analysis is an efficient approach for software assurance. It is indicated that its most effective usage is to perform analysis in an interactive way through the software development process, which has a high performance requirement. This paper concentrates on rule-based static analysis tools and proposes an optimized rule-checking algorithm. Our technique improves the performance of static analysis tools by filtering vulnerability rules in terms of characteristic objects before checking source files. Since a source file always contains vulnerabilities of a small part of rules rather than all, our approach may achieve better performance. To investigate our technique's feasibility and effectiveness, we implemented it in an open source static analysis tool called PMD and used it to conduct experiments. Experimental results show that our approach can obtain an average performance promotion of 28.7% compared with the original PMD. While our approach is effective and precise in detecting vulnerabilities, there is no side effect.

Keywords: Rule-based static analysis; Software quality; Software validation; Performance improvement
http://dx.doi.org/10.1631/FITEE.1500379 **CLC number:** TP311

1 Introduction

Static analysis is one of the most important techniques for software assurance. It can be used to find potential vulnerabilities as early as in the software implementation phase by scanning application programs' source code, byte code, binary code, or other artifacts rather than running the executable files. In recent years, many tools have been developed to automatically find bugs in program source code, using techniques such as syntactic pattern matching (Atkinson and Griswold, 2006), data flow analysis, type systems, model checking, and theorem proving (Rutar *et al.*, 2004). As pointed out by Hovemeyer and Pugh (2004), integrating bug-finding tools into the development process is an important direction for future research. These tools not only are powerful but also possess the feature to be used in an interactive way through the software development process. For instance, Coverity and Klocwork (Emanuelsson and Nilsson, 2008) are two powerful software assurance tools, both of which have plug-in versions for Eclipse IDE. However, the plug-in versions are always more

‡ Corresponding author

* Project supported by the National High-Tech R&D Program (863) of China (No. 2013AA12A202), the National Natural Science Foundation of China (Nos. 61172173, 41501505, and 61502205), the Natural Science Foundation of Hubei Province, China (No. 2014CFB779), and the Youths Science Foundation of Wuhan Institute of Technology (No. K201546)

ORCID: Deng CHEN, http://orcid.org/0000-0001-6359-801X lightweight tools, which just carry out shallow analysis due to high performance requirements.

To deal with the performance issue, many techniques have been proposed, such as the incremental analysis technique and distributed technology (Hovemeyer and Pugh, 2004). Incremental analysis automatically infers what parts of source code have to be reanalyzed after the code has been modified. This approach typically reduces the analysis time substantially, but may, of course, imply a complete reanalysis in the worst case (Emanuelsson and Nilsson, 2008). Distributed technology is a general approach to cope with efficiency issues, but building a robust distributed system is a complicated and complex task. Apart from the above techniques, some other techniques exist. However, many of them have side effects on the precision of static analysis tools.

In this paper, we aim to improve the performance of static analysis tools without side effects on the detection capability and precision. Our approach is designed for rule-based static analysis tools. A rule-based static analysis tool detects vulnerabilities by checking program source code or other artifacts against vulnerability rules lexically or syntactically. We focus on this particular type of tool for several reasons: (1) This kind of tool is used extensively and many recently developed tools are of the kind, such as FindBugs (Hovemeyer and Pugh, 2004; 2007; Ayewah *et al.*, 2007; Araújo *et al.*, 2011), PMD (Helmick, 2007; Plosch *et al.*, 2008; Araújo *et al.*, 2011), and CheckStyle (Loveland, 2009). (2) This kind of tool is able to detect many kinds of vulnerabilities only if the corresponding rules are included. (3) These tools have some unique characteristics which make it possible to improve their performance by novel approaches rather than the above traditional ones.

Rule-based static analysis tools always work in a manner as follows. First, they translate program source code into an intermediate form, such as the abstract syntax tree (AST). Then, they detect vulnerabilities by checking the AST against vulnerability rules. Finally, they report vulnerabilities in different formats. The second phase is crucial for rule-based static analysis tools, which is always performed using a rule-checking algorithm. To date, most rule-checking algorithms have been designed based on the visitor pattern; that is, each AST node is matched with all vulnerability rules using a depth- or breadth-first tree-searching approach. The problem with this approach is that a large number of unnecessary rules are considered, which will result in much runtime overhead. Based on this observation, we propose an optimized rule-checking algorithm. Given a source file *f* that is required to be validated, prior to checking the file against vulnerability rules, our algorithm first filters out unnecessary rules regarding *f* according to their characteristic objects (which is defined in later sections). We say a vulnerability rule *r* is unnecessary regarding a source file *f*, if it is impossible for *f* to contain violations of *r*. Our approach should be effective in improving the performance of static analysis tools, because a source file always contains violations of a small part of rules rather than all. Additionally, since the abandoned rules are unnecessary, no side effects will be caused to the detection capability and precision.

We filter vulnerability rules in terms of their characteristic objects. The characteristic objects of a vulnerability rule *r* is a set of objects that form a necessary condition for a source file containing violations of *r*. If a source file *f* contains violations of *r*, *f* should contain characteristic objects of *r*, but not vice versa. In this work, we use the classes included in a vulnerability rule to form the characteristic objects. In addition, a prefix notation (http://en.wikipedia.org/wiki/Polish_notation) CObject expression is designed to describe characteristic objects and relationships among them. Relying on the evaluation results of CObject expressions, we filter vulnerability rules effectively.

To investigate our technique's feasibility and effectiveness, we implemented it in a prototype tool EPMD, which was developed from an open source static analysis tool called PMD, and used the tool to perform experiments. The experimental results show that our approach can improve the performance of static analysis tools effectively with an average promotion of 28.7%. The earlier conference version of this paper appeared in Chen *et al.* (2014).

The contributions of this paper include: (1) an optimized rule-checking algorithm that can improve the performance of rule-based static analysis tools; (2) a formal language that is used to describe characteristic objects of rules; (3) a prototype tool EPMD that implements our technique; (4) an experiment that evaluates the effectiveness of our technique.

2 Background

2.1 Static and dynamic analysis techniques

Generally, there are two categories of automatic bug-finding approaches: program static analysis techniques and program dynamic analysis techniques.

Program static analysis techniques (Rutar *et al.*, 2004; Hovemeyer and Pugh, 2004; 2007; Atkinson and Griswold, 2006; Ayewah *et al.*, 2007; Helmick, 2007; Emanuelsson and Nilsson, 2008; Plosch *et al.*, 2008; Araújo *et al.*, 2011) take program source code, byte code, binary code, or other artifacts as input and find bugs using techniques such as syntactic pattern matching, data flow analysis, type systems, and model checking. Program dynamic analysis techniques do not require program source code as input. They find bugs from program runtime information, which is collected by running application programs. The two kinds of techniques have their own advantages and disadvantages. Static analysis approaches can find bugs as early as in the software implementation phase. Additionally, they can be efficient and highly automated. Furthermore, they can cover all the program paths, and latent specifications provide some benefits such as inferring relationships among variables according to their naming conventions (Engler *et al.*, 2001). However, they should deal with some intractable problems: infeasible paths, complicated data structures, pointer aliasing, etc. Compared with static analysis techniques, dynamic analysis techniques (Bounimova *et al.*, 2013; Haydar *et al.*, 2013; Reinbacher *et al.*, 2014) can be used more extensively, especially when source codes are unavailable. Additionally, since they analyze programs based on runtime information, a higher precision may be achieved. Apart from that, the tricky problems with static analysis can be avoided. However, dynamic analysis techniques have the following drawbacks: (1) Since they require running applications, they are less efficient than static analysis techniques. (2) Before running applications, a lot of configuration work should be done, which makes these approaches difficult to automate. (3) They always employ instrumentation techniques to extract runtime information, which will incur much runtime overhead. (4) The effect of these approaches largely depends on the quality of input test cases.

2.2 Rule-based static analysis tools

In this work, we focus on static analysis techniques. Although these techniques have some weaknesses, many academic and commercial static analysis tools have still been developed and used extensively in practice, such as FindBugs (Hovemeyer and Pugh, 2004; 2007; Ayewah *et al.*, 2007; Araújo *et al.*, 2011), PMD (Helmick, 2007; Plosch *et al.*, 2008; Araújo *et al.*, 2011), CheckStyle (Loveland, 2009), PREFix (Rajamani, 2006; Ball, 2008), PREFast (Rajamani, 2006; Ball, 2008), and Metal (Rajamani, 2006). Most of these tools work with a library of vulnerability rules. A vulnerability rule is a program specification (protocol) that the source code is expected to satisfy. Violations of vulnerability rules will cause program errors. Additionally, the number of vulnerability rules is an important indicator of the detection capability of a static analysis tool. In this work, we call the kind of static analysis tools with a library of vulnerability rules 'rule-based static analysis tools'. As illustrated in Fig. 1, rule-based static analysis tools typically consist of four parts: compiler front end, vulnerability rules library, rules checking engine, and vulnerability reporter. Compiler front end is responsible for translating program source code into an intermediate form, such as AST. The vulnerability rules library contains a set of vulnerability rules. The function of the vulnerability reporter is to output the discovered vulnerabilities in different formats. The rules checking engine finds bugs by

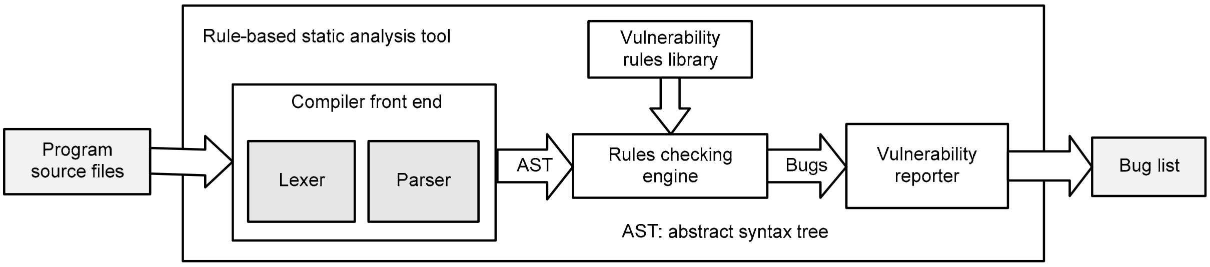


Fig. 1 Composition of rule-based static analysis tools

checking AST nodes against vulnerability rules. The rule-checking algorithm that it employs is a decisive factor of the performance of rule-based static analysis tools, especially when a large number of vulnerability rules are included.

2.3 Rule-checking algorithm

To find bugs, rule-based static analysis tools take program source files as input and translate them into ASTs using compiler front end. Then, the rules checking engine validates programs by checking ASTs against vulnerability rules.

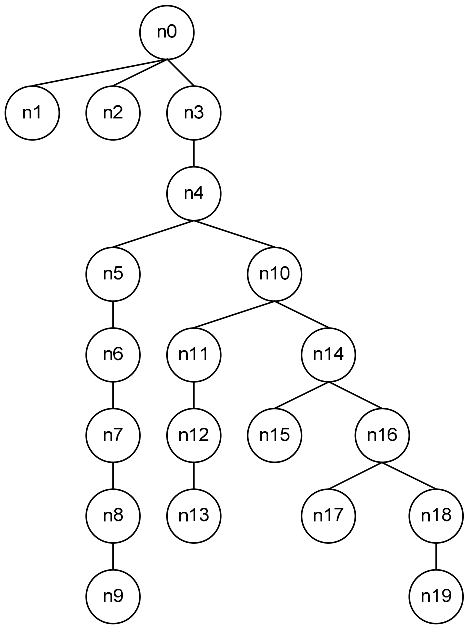
Fig. 2 gives a Java program, a compressed AST of which is shown in Fig. 3 with some trivial nodes neglected for simplicity. The rule-checking algorithm traverses the AST top down using a depth- or breadth-first tree search approach and matches AST nodes with vulnerability rules. If a node *n* matches a rule *r*, *n* and the surrounding nodes above and below *n* will be checked against *r* and violations will be reported. The whole AST and all vulnerability rules should be processed, because an AST node may match multiple vulnerability rules. As we can see, although the program shown in Fig. 2 has only five code lines, its AST is far from trivial and has more than 20 nodes. If the vulnerability rules library is large, the rule-checking algorithm must be time consuming.

```
package pmd_test;

import java.text.SimpleDateFormat;

public class Pmd_Test {
   public static void main(String[] args) {
      SimpleDateFormat sdf = new
         SimpleDateFormat("pattern");
   }
}
```

Fig. 2 Example of a Java program

2.4 Rule description languages

Vulnerability rules are described using various kinds of rule description languages, such as PQL (Jarzabek, 1998; Martin *et al.*, 2005), Datalog (Whaley *et al.*, 2005; Hajiyev *et al.*, 2006; Alpuente *et al.*, 2009; Zook *et al.*, 2009), and XPath expression (Panchenko *et al.*, 2010; 2011). Fig. 4 illustrates a vulnerability rule MDBAndSessionBeanNamingConvention (MSBNC) described using an XPath expression, which is excerpted from the rule library of PMD. It is designed to check the EJB specification in which the name of any class implementing MessageDrivenBean or SessionBean interface should be suffixed by ‘Bean’. Based on rule description languages, we can compose various kinds of vulnerability rules to extend the detection capability of rule-based static analysis tools.



(a)

TAG	TYPE	VALUE
n0	CompilationUnit	
n1	PackageDeclaration	pmd_test
n2	ImportDeclaration	java.text.SimpleDateFormat
n3	ClassOrInterfaceDeclaration	Pmd_Test
n4	MethodDeclaration	ResultType:void
n5	MethodDeclarator	main
n6	FormalParameters	Count:1
n7	FormalParameter	args
n8	Type	
n9	ClassOrInterfaceType	string
n10	LocalVariableDeclaration	
n11	Type	
n12	ReferenceType	
n13	ClassOrInterfaceType	SimpleDateFormat
n14	VariableDeclarator	
n15	VariableDeclaratorId	sdf
n16	AllocationExpression	
n17	ClassOrInterfaceType	SimpleDateFormat
n18	Arguments	Count:1
n19	Literal	"pattern"

(b)

Fig. 3 A compressed abstract syntax tree (AST) of the Java program shown in Fig. 2: (a) AST; (b) information of AST nodes

```
<![CDATA[
// TypeDeclaration/ClassOrInterfaceDeclaration
[
   (
      (./ImplementsList/ClassOrInterfaceType
      [ends-with(@Image,'SessionBean')]) or
      (./ImplementsList/ClassOrInterfaceType

[ends-with(@Image,'MessageDrivenBean')])
   )
   and not (ends-with(@Image,'Bean'))
]]]>
```

Fig. 4 Vulnerability rule MSBNC excerpted from PMD

3 Motivating example

Although many static analysis tools (Loveland, 2009; http://findbugs.sourceforge.net; http://pmd.sourceforge.net) have improved the rule-checking algorithm to some extent in implementation, the effect is not significant. In this section, we present an extremely bad case that our technique intends to deal with.

Fig. 5 shows the XPath expression of a vulnerability rule SimpleDateFormatNeedsLocale (SDFNL). It is excerpted from the rule library of PMD. The rule is designed to detect a misuse of class SimpleDateFormat from JDK. SimpleDateFormat is a concrete class for formatting and parsing dates in a locale-sensitive manner. One of its constructors has two parameters. The first one is a pattern describing the date and time format. The second one is the locale. A common error in instantiating objects from the class is shown at line 7 of the program illustrated in Fig. 2, where only the pattern is specified but the locale is neglected.

```
<![CDATA[
// AllocationExpression
[ClassOrInterface-
Type[@Image='SimpleDateFormat']]
[Arguments[@ArgumentCount=1]]
]]>
```

Fig. 5 Vulnerability rule SDFNL excerpted from PMD

Let us consider checking the program shown in Fig. 6 against the above rule SDFNL. By manual inspection, we can confirm that no violations of the rule exist in the program, because the program has never used class SimpleDateFormat. However, the rule-checking algorithm will also traverse the whole AST of the program and match the AST nodes with all vulnerability rules until no match is found. Virtually, this situation is ubiquitous in static analysis and results in enormous time overhead. To mitigate the problem, we propose an optimized rule-checking algorithm, which can deal with the situation effectively. Details of our technique will be discussed in Section 4.

```
package pmd_test;

import java.util.Collection;
import java.util.ArrayList;

public class Pmd_Test {
   public static void main(String[] args) {
      Collection c = new ArrayList();
      Integer obj = new Integer(1);
      c.add(obj);

      Integer[] a = (Integer [])c.toArray();
   }
}
```

Fig. 6 Example of Java program

4 Our technique

In this section, we present our technique of checking source files against vulnerability rules. Our technique can handle application programs written in most of mainstream programming languages (e.g., Java, C/C++, and C#). In this section, we take Java programs as an example to demonstrate its working principle. We first provide an intuitive description of our technique and then discuss its main characteristics in detail.

4.1 General approach

The rule-checking algorithm takes ASTs and vulnerability rules as input. To find bugs in a source file, it traverses the AST translated from the source file top down using a depth- or breadth-first tree-searching approach. Each AST node is matched with all vulnerability rules. When a match (n, r) is found, where n and r are AST node and vulnerability rule, respectively, node n and its surrounding nodes will be checked against rule r, and violations will be reported. The above rule-checking algorithm can be reduced to a general searching problem: let N and R be the sets of AST nodes and vulnerability rules, respectively. The rule-checking algorithm aims at finding a match (n, r) from the Cartesian product $N×R$.

Generally, the time overhead of the rule-checking algorithm is proportional to $|N\times R|$, where '$|\cdot|$' denotes the cardinality of a set. Since

$$|N\times R|=|N|\times|R|,$$

we can improve the performance of rule-checking algorithms by reducing $|N|$ or $|R|$, that is, cutting down the number of AST nodes or vulnerability rules used for matching. In this work, we optimize the rule-checking algorithm by filtering vulnerability rules.

We filter vulnerability rules according to their characteristic objects. The characteristic objects of a vulnerability rule r is a set of objects that form a necessary condition for a source file containing violations of r. In other words, if a source file f contains violations of r, f should contain characteristic objects of r, but not vice versa. In this work, we use the classes included in a vulnerability rule to form the characteristic objects. Our technique is based on the observation that a vulnerability rule is always dedicated to some particular classes. Take the rule MSBNC illustrated in Fig. 4 as an example; it aims to detect violations concerning class MessageDrivenBean and SessionBean. We can conclude that a source file must be free of violations of the rule if neither of the classes has been used in the file; that is, the classes form a necessary condition of rule MSBNC.

Characteristic objects may have various kinds of logical relationships. Consider the vulnerability rule shown in Fig. 4. A disjunction relationship exists between its characteristic objects MessageDrivenBean and SessionBean. To deal with the situation, we describe characteristic objects using a prefix notation CObject expression, which is able to express complicated logical relationships. Given a source file f and a set of vulnerability rules R, our rule-checking algorithm validates f against R based on the following approach: for each rule $r\in R$, COBj(r) denotes the CObject expression of r. We evaluate CObj(r) against f and achieve a Boolean result p. If p equals *true*, we add r to a set of vulnerability rules R'; otherwise, we abandon rule r. Finally, we validate the source file f against rule set R' rather than R. The outline of our rule-checking algorithm is presented in Algorithm 1, where evaluate(CObj(r), f) denotes the evaluation of CObj(r) against file f.

Algorithm 1 Our rule-checking algorithm

```
Input: f, a source file; R, the library of vulnerability rules
Methods:
for each rule r in R do
    P←evaluate(CObj(r), f)
    if P equals true then
        Add r to a set of vulnerability rules R′
    end if
end for
Validate f against rule set R′
```

Note that the evaluation of CObj(r) against source file f may be time consuming especially when the file is large. To address the problem, we approximately evaluate CObj(r) against f by evaluate(CObj(r), P_f), where P_f is the set of *package* statements at the beginning of file f. The *package* statements are used to import a package of objects into a source file, supported by most of mainstream languages, such as the *import* statements in Java, *include* statements in C/C++, and *using* statements in C#. Since the *package* statements are always at the beginning of a source file, they can be extracted conveniently. Additionally, the number of *package* statements of a source file is much smaller than that of the code lines in the source file. Consequently, our approximate evaluation of CObj(r) against P_f should be more efficient than that against f.

4.2 CObject expression

To describe characteristic objects as well as relationships among them, we propose a prefix notation CObject expression.

A CObject expression is composed of characteristic objects, parameters, commas, parentheses, and operator symbols. The recursive definition of CObject expression is as follows. Given a characteristic object λ and two CObject expressions R and S, we have the following CObject expressions:

exist(λ, F) denotes the *existence* operation that determines whether the characteristic object λ has been used in a source file F. The source file F is a parameter of the *exist* operator. When evaluating CObject expressions, we will apply a source file to the *exist* operator; that is, we substitute F with an argument. For the convenience of notation, we always omit parameter F in CObject expressions.

neg(R) denotes the *negation* operation that reverses the meaning of a Boolean operand.

and(*R*, *S*) denotes the *conjunction* operation that performs a logical conjunction on two Boolean operands.

or(*R*, *S*) denotes the *disjunction* operation that performs a logical disjunction on two Boolean operands.

The CObject expression is compact and easy to use. Additionally, in our formal language, operators can be nested within each other. This makes our language powerful enough to express complex vulnerability rules. For instance, the characteristic objects of the vulnerability rule MSBNC shown in Fig. 4 can be expressed using a CObject expression as follows:

$$or(exist(\text{SessionBean}),\ exist(\text{MessageDrivenBean})),$$

which means that violations of rule MSBNC may exist in a source file *f*, if either of the classes MessageDrivenBean and SessionBean has been used in *f*. Note that we omit the prefix javax.ejb of the fully qualified names javax.ejb.MessageDriven Bean and javax.ejb.SessionBean for notational convenience.

Given a vulnerability rule *r*, the evaluation result of CObj(*r*) against a source file *f* is a Boolean value *p*. If *p* equals *true*, it indicates that violations of *r* may exist in *f*; otherwise, *f* should be free of violations of rule *r*. Based on the evaluation results of CObject expressions, we can filter the vulnerability rules effectively.

To examine the expressing capability of CObject expressions, we inspect all rules of PMD, and no exceptions are found. The CObject expression may be limited for some unknown rules (e.g., rules whose characteristic objects have an if-then relationship), but we believe it is applicable in most of cases. For further understanding of the CObject expression, we present its EBNF grammar specification in Fig. 7.

4.3 Evaluating CObject expressions

In this subsection, we introduce our technique of evaluating CObject expressions.

Let *r* be a vulnerability rule. Before evaluating CObj(*r*) against a source file *f*, we should apply *f* to CObj(*r*); that is, we have the following transformation:

$$\text{evaluate}(\text{CObj}(r), f) \Rightarrow \text{evaluate}(\text{CObj}(r)(f)).$$

CObj(*r*)(*f*) is also a prefix notation, which is achieved by substituting the parameter of *exist* operators in CObj(*r*) with argument *f*. Taking the CObject expression of the vulnerability rule MSBNC shown in Section 4.2 as an example, we have the following results:

$$\begin{aligned}\text{CObj}(\text{MSBNC})(f) \Rightarrow\ & or(exist(\text{SessionBean}, f),\\ & exist(\text{MessageDrivenBean}, f)).\end{aligned}$$

startexpr	:	*expr*;
expr	:	*binexpr* | *unaryexpr* | *atomexpr*;
binexpr	:	BINOP LPAREN *expr* COMMA *expr* RPAREN;
unaryexpr	:	UNARYOP LPAREN *expr* RPAREN;
atomexpr	:	EXISTOP LPAREN *object* COMMA 'F' RPAREN;
object	:	(*obname* DOT)* *obname*;
obname	:	(LETTER | UNDERLINE)(LETTER | NUMBER | UNDERLINE)*;
BINOP	:	**'and'** | **'or'**;
UNARYOP	:	**'neg'**;
EXISTOP	:	**'exist'**;
DOT	:	'.';
LETTER	:	'a'...'z' | 'A'...'Z';
NUMBER	:	'0'...'9';
UNDERLINE	:	'_';
LPAREN	:	'(';
RPAREN	:	')';
COMMA	:	',';

Fig. 7 EBNF grammar specification of the CObject expression

After that, we evaluate CObj(*r*)(*f*) using the classic stack-based approach (http://en.wikipedia.org/wiki/Polish_notation), which is dedicated to prefix notations. The method scans CObj(*r*)(*f*) from right to left. For each encountered token η, the following actions are conducted according to the token type:

1. If η is an operand, push it into a stack *T*.

2. If η is an operator, perform corresponding operations on operands popped from the top of *T* and push the results back into *T*.

3. If η is a separator, such as commas and parentheses, discard it.

Once all the symbols have been processed, a Boolean value will be left in *T*, which is the evaluation result. The stack-based approach is efficient and only a stack with the maximum length of *n* is employed, where *n* is the length of CObj(*r*)(*f*). Both the time and space complexities of the stack-based approach are $O(n)$.

4.4 Existence operation

Note that a CObject expression subsumes four kinds of operators: *neg*, *and*, *or*, and *exist*. The former three are logical operators. As to the last one, a brute-force approach to carry out the operation is as follows: given a characteristic object λ and a source file f, we evaluate *exist*(λ, f) by searching the AST of f using a depth- or breadth-first tree-searching approach. Each AST node is matched with λ. A result of Boolean value *true* is achieved if any match is found. Otherwise, we have a result of *false*.

Obviously, the above approach may cause significant time overhead if the source file f is large. To mitigate the problem, we approximately evaluate *exist*(λ, f) in terms of P_f rather than f, where P_f is the set of *package* statements at the beginning of f. In other words, we have the following transformation:

$$\text{evaluate}(exist(\lambda, f)) \Rightarrow \text{evaluate}(exist(\lambda, P_f)).$$

Our method is based on a common programming practice: to use a class in a source file, programmers will first import it using package statements at the beginning of the source file. Taking the Java class SessionBean as an example, it may be imported using one of the following statements: (1) import javax.ejb. SessionBean; (2) import javax.ejb.*.

The first statement imports exactly one class into a source file, i.e., SessionBean. The second statement imports classes in the package javax.ejb on demand. The *package* statement is supported by many programming languages, such as the *import* statement in Java programs, *include* statement in C/C++ programs, and *using* statement in C# programs. Since *package* statements are at the beginning of source files, they can be recognized conveniently. On the other hand, the number of *package* statements in a source file is trivial compared to that of the code lines. Consequently, our technique should be efficient.

Note that some *package* statements may not be included in ASTs after a source file is compiled, such as the preprocessor statement *include* in C/C++ programs. In this situation, our rule-checking algorithm should take ASTs and *package* statements as input.

Also, note that our technique of evaluating the *existence* operation may not work properly in some special cases. Taking the *import* statement in Java programs as an example, the programming practice that a class should be imported into a source file before its usage may be violated in the following situations: (1) Classes within package java.lang can be used in a source file without *import* statements. (2) Redundant *import* statements may exist in source files. (3) We cannot ascertain the existence of a specific object according to *import-on-demand* statements (e.g., import javax.ejb.*). (4) We can use an object via its fully qualified name without *import* statements. To deal with case 1, we can evaluate the *existence* operation against the whole source file f when the characteristic object λ belongs to the package java.lang. As to cases 2 and 3, our approximate evaluation method will give a wrong result *true*, even if a characteristic object has not been used in a source file. Finally, the failure to filter out unnecessary rules may result. The last case is worse, because it may incur false negatives. In practice, we should avoid this situation when applying our technique. Although the above cases may discount the usability of our technique, they are rare in programs coded with good programming practices (some program analysis tools even take redundant *import* statements as errors). Therefore, our technique is applicable in most cases.

The above discussions are dedicated to Java programs. As to other programming languages, some limitations may not exist. For example, the above cases 1 and 4 do not exist in C/C++ programs, which makes our technique more useful.

4.5 An evaluation example

As an example, we evaluate the CObject expression of rule MSBNC shown in Fig. 4 against an input program ρ illustrated in Fig. 6. For simplicity, we omit the prefix javax.ejb of the fully qualified names javax.ejb.MessageDrivenBean and javax.ejb. SessionBean. The CObject expression after applying program ρ is as follows:

$$or(exist(\text{SessionBean}, \rho), exist(\text{MessageDrivenBean}, \rho)).$$

The evaluation process is presented in Table 1. As we can see, the evaluation result is Boolean value *false*, which is consistent with our manual inspection. According to the evaluation results, we have the conclusion that it is unnecessary to check program ρ against rule MSBNC.

Table 1 Evaluation example

Step	Token (η)	Stack* (T)	Action
1	)		Discard
2	)		Discard
3	ρ	ρ	Push ρ into stack T
4	,	ρ	Discard
5	MessageDrivenBean	ρ, MessageDrivenBean	Push MessageDrivenBean into T
6	(	ρ, MessageDrivenBean	Discard
7	*exist*	*false*	Pop operands MessageDrivenBean and ρ Evaluate expression *exist*(MessageDrivenBean, ρ)→*false* Push evaluation result *false* into stack T
8	,	*false*	Discard
9	)	*false*	Discard
10	ρ	*false*, ρ	Push ρ into stack T
11	,	*false*, ρ	Discard
12	SessionBean	*false*, ρ, SessionBean	Push SessionBean into stack T
13	(	*false*, ρ, SessionBean	Discard
14	*exist*	*false*, *false*	Pop operands SessionBean and ρ Evaluate expression *exist*(SessionBean, ρ)→*false* Push evaluation result *false* into stack T
15	(	*false*, *false*	Discard
16	*or*	*false*	Pop operands *false* and *false* Evaluate expression *false*∨*false*→*false* Push evaluation result *false* into stack T
17			Pop and output evaluation result *false*

* The left and right ends are the bottom and top of stack T, respectively

5 Experiments

To evaluate our technique, we implemented it in a prototype tool called EPMD, which was developed from the open source static analysis tool PMD. After that, we evaluated the effectiveness of our technique through a comparison test based on EPMD. In this section, we first introduce our prototype tool EPMD. Then, we present the subject programs used in our experiment. Next, we elaborate the comparison test based on EPMD and present experimental results. Finally, we discuss our technique's runtime overhead and limitations.

5.1 Prototype tool EPMD

We implemented our technique in an open source static analysis tool PMD and called it the extended version (EPMD). PMD is designed to find mistakes and smells in Java programs, such as unused variables, empty catch blocks, and unnecessary object creation (http://pmd.sourceforge.net). It supports two kinds of vulnerability rules: Java classes and XPath expressions. Our primary extensions to PMD are: (1) implementation of a module that evaluates CObject expressions; (2) addition of CObject expressions to the vulnerability rules of PMD.

To evaluate CObject expressions, we constructed a lexer and parser, which are used to recognize tokens and validate the syntax of CObject expressions, respectively. The lexer and parser were constructed with the help of ANTLR (Schaps, 1999; Bovet and Parr, 2008; Liu *et al.*, 2008; Parr and Fisher, 2011), which is a language tool. After that, EPMD evaluates CObject expressions using the stack-based approach elaborated in Section 4.

We added CObject expressions to the vulnerability rules of PMD by inserting a property CObjectExpr to their XML descriptions, which is shown below:

```
<property name="CObjectExpr" type="String"
description="rule's CObject expression"
value="exist(java.ejb.EJBHome)"/>
```

Note that not all rules have CObject expressions (which will be discussed in Section 5.5). Our technique will not work for rules without the CObjectExpr property.

5.2 Subjects

We adopted six large-scale open source applications in our experiments (Table 2). These applications were selected based on the following considerations: (1) applications with available source code, (2) applications of a large code base, and (3) applications coming from various application domains. The second requirement is important for our evaluation, because the performance improvement gained by the removal of one unnecessary rule is trivial. To achieve accurate and salient results, we should perform experiments on large-scale applications, where a large number of unnecessary rules may be pruned away. Furthermore, to avoid biases, we selected subject programs from various application domains. Additionally, they were of different sizes spanning from 73 kilo lines of code (KLoC) to 2675 KLoC (we counted code lines only; comment lines and blank lines were excluded).

Table 2 Subject programs used in our experiments

Subject program	Description	Number of source files	KLoC
Eclipse 4.2	Software development environment	20 602	2675
ElasticSearch v0.19.9	Distributed search engine	2581	230
PMD 5.0	Source code analyzer	945	73
Tomcat 6.0.35	Web server and servlet container	1157	172
SQuirreL SQL Client 3.4.0	Java SQL client	2890	253
JBoss 6.0.0.Final	Application server	6397	494

KLoC: kilo lines of code

5.3 Effectiveness of our technique

To investigate the effectiveness of our technique, we analyzed subject applications using EPMD with rule sets RS and RS*, respectively. Rule sets RS and RS* comprise 45 vulnerability rules excerpted from PMD (part of vulnerability rules can be found in Appendix). These two rule sets are nearly the same, except that rules in RS* have an additional property CObjectExpr. Since our technique will not work for rules without the property CObjectExpr, we can investigate the effectiveness of our technique through a comparison test with rule sets RS and RS*. To achieve accurate results, we ran EPMD five times for each group of subjects and rule sets. The platform used in our experiment is an Intel® Core™ i3-2100 3.1 GHz machine with 3 GB of memory running Windows XP. Table 3 presents the average execution time of EPMD running with rule sets RS and RS*.

We can observe from Table 3 that the execution time of EPMD with rule set RS* is generally less than that with RS for all subject applications. The least ratio of reduction is 15.6% when testing on Eclipse. The ultimate ratio is 45.3% with PMD. We have an average time reduction ratio of 28.7%. A side effect that may be caused by our technique is the increase of false positives and false negatives. We compared the bug reports generated by EPMD in each group and found that the bugs reported by EPMD running with rule sets RS and RS* were exactly the same. This result suggests that the fully qualified names of classes that may cause false negatives to our technique are rarely used in Java programs.

In our experiment, we validated six real-world programs against 45 vulnerability rules. The experimental results strongly suggest that our technique is effective in improving the performance of static analysis tools without side effects. Moreover, better results may be achieved if more vulnerability rules are considered, which we will further investigate in the future.

5.4 Runtime overhead

The primary runtime overhead of our technique is introduced in the phase of filtering rules (Section 4). To filter rules, we evaluated CObject expressions based on the approach applied to prefix notations. Different actions were carried out according to the type of operator, among which the *existence* operation is crucial. Let L and M be the counts of tokens of a CObject expression and *package* statements of a source file, respectively. Then the time complexity of the stack-based evaluation approach is $O(L)$. The

Table 3 Results of our comparison test with EPMD being run five times for each group of subjects and rule sets

Subject program	Rule set	Execution time (s)					AVG (s)	RDT (s)	RDT ratio
		1	2	3	4	5			
Eclipse	RS	469.8	437.4	444.0	438.9	467.1	451.4	70.4	15.6%
	RS*	380.3	380.2	380.3	382.9	381.6	381.1		
ElasticSearch	RS	40.7	36.9	37.1	36.7	39.7	38.2	6.6	17.3%
	RS*	31.5	31.7	31.7	31.6	31.5	31.6		
PMD	RS	9.5	6.6	6.7	6.6	8.6	7.6	3.4	45.3%
	RS*	4.1	4.2	4.1	4.2	4.2	4.2		
Tomcat	RS	15.5	14.8	14.6	14.6	15.2	14.9	5.4	36.4%
	RS*	9.6	9.5	9.3	9.6	9.5	9.5		
SQuirreL SQL Client	RS	45.2	31.2	31.4	48.1	31.1	37.4	11.1	29.6%
	RS*	26.4	26.5	26.2	25.8	26.8	26.3		
Jboss	RS	57.8	49.0	49.2	66.3	49.6	54.4	15.4	28.2%
	RS*	38.8	39.2	39.0	39.0	39.1	39.0		

AVG: average execution time of EMPD; RDT: average reduction of execution time

existence operation has a time complexity of $O(M)$ because it consists mainly of a loop iterating through all *package* statements. Therefore, the time complexity of filtering rules is $O(L×M)$. On the other hand, it is quite reasonable to assume L to be a constant because, in general, the CObject expressions of vulnerability rules are of limited length. Additionally, it is well known that the number of *package* statements is trivial relative to the number of lines of code in a source file. Thus, we believe that the runtime overhead incurred by our technique is within an acceptable range.

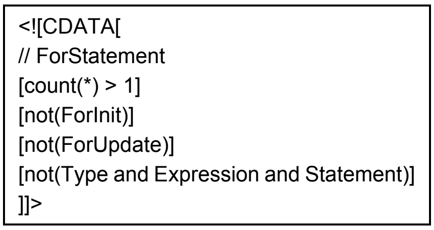
5.5 Limitations

Although our technique is effective in improving the performance of static analysis tools, it has the following limitations:

1. As discussed in Section 4, our approximate evaluation approach of the *existence* operation based on *package* statements may not work properly for programs coded with poor programming practices. The requirement for high-quality programs may limit the application scope of our technique.

2. In this work, we compose the CObject expressions of vulnerability rules manually. An automatic approach to derive the CObject expressions from the description of vulnerability rules may make our technique more useful.

3. There exist some rules that have no characteristic objects. Consider the rule ForLoopShouldBeWhileLoop illustrated in Fig. 8. It is designed to detect *for* loops that should be simplified to *while* loops. As we can see, this rule does not possess any characteristic objects but for a keyword *for*. Our technique is unable to deal with this situation.



```
<![CDATA[
// ForStatement
[count(*) > 1]
[not(ForInit)]
[not(ForUpdate)]
[not(Type and Expression and Statement)]
]]>
```

Fig. 8 Rule ForLoopShouldBeWhileLoop excerpted from PMD

Although our technique has some limitations, as above, it is applicable in most of circumstances. The evaluation results showed that our technique is effective in improving the performance of static analysis tools.

6 Related work

Though some state-of-the-art static analysis tools are efficient enough to satisfy most applications, performance is an eternal topic, and it is still a key challenge to integrate bug-finding tools into the development process. To deal with the performance issue, researchers have proposed various kinds of techniques. In this section, we focus on several typical techniques.

Incremental analysis (Emanuelsson and Nilsson, 2008) can automatically infer which parts of source code have to be reanalyzed after the code has been modified. This approach typically reduces the analysis time substantially, especially for a frequent interactive use of static analysis tools in the development process. However, it is difficult to accurately identify the associated code when programs have been changed; in the worst case, a complete reanalysis will be performed.

A commonly used approach to improve the performance of static analysis tools is to perform a less deep analysis, such as GREP (Atkinson and Griswold, 2006; Emanuelsson and Nilsson, 2008). It performs lexical analysis only. However, these tools always have a high rate of false positives, which discounts their usability.

The technique used by PMD is similar to our approach, which improves the rule-checking algorithm by filtering rules leveraging a data structure Rule Chain. A Rule Chain divides all AST nodes into different categories, where nodes in each category have the same programming language and node type. With the help of the Rule Chain, PMD checks AST nodes against vulnerability rules with the same programming language and type rather than all. This approach is effective in improving the efficiency of rule-checking algorithms. However, it cannot exclude all unnecessary vulnerability rules because such rules may have the same programming language and type as an AST node.

Many static analysis tools, such as FindBugs (http://findbugs.sourceforge.net) and CheckStyle (Loveland, 2009), provide the functionality for users to select vulnerability rules before checking a program. This approach is most similar to our technique. Both of them aim to improve performance by filtering rules. However, our technique can filter rules automatically.

7 Conclusions and future work

Performance is a key challenge of integrating bug-finding tools into a development process. In this paper, we proposed an optimized rule-checking algorithm that can improve the performance of static analysis tools without side effects on their detection capability and precision. Our rule-checking algorithm filters unnecessary vulnerability rules automatically by evaluating their CObject expressions. Although we discussed our technique based on Java programs and vulnerability rules described using XPath expressions, our method is a general approach that is applicable for most of mainstream programming languages and other types of vulnerability rules. To evaluate the effectiveness of our technique, we implemented it in an open source static analysis tool PMD and called it the extended version (EPMD). Relying on EPMD, we performed a comparison test and found that our technique can achieve an average performance promotion of 28.7%. Additionally, few false positives and false negatives were caused.

As discussed in Section 5, our technique has some limitations, which may discount its usability. In the future, we will mitigate these problems and make our technique more useful. For example, we may implement our technique for other programming languages, where some limitations regarding Java programs may not exist. Additionally, we may further promote the ease of use of our technique by deriving CObject expressions from descriptions of vulnerability rules automatically. This task can be performed by leveraging a syntax transducer, which parses XPath expressions of vulnerability rules and translates them to CObject expressions.

References

Alpuente, M., Feliú, M.A., Joubert, C., *et al.*, 2009. Using Datalog and Boolean equation systems for program analysis. 13th Int. Workshop on Formal Methods for Industrial Critical Systems, p.215-231. http://dx.doi.org/10.1007/978-3-642-03240-0_18

Araújo, J.E.M., Souza, S., Valente, M.T., 2011. Study on the relevance of the warnings reported by Java bug-finding tools. *IET Softw.*, **5**(4):366-374. http://dx.doi.org/10.1049/iet-sen.2009.0083

Atkinson, D.C., Griswold, W.G., 2006. Effective pattern matching of source code using abstract syntax patterns. *Softw.-Pract. Exp.*, **36**(4):413-447. http://dx.doi.org/10.1002/spe.704

Ayewah, N., Pugh, W., Morgenthaler, J.D., *et al.*, 2007. Evaluating static analysis defect warnings on production software. ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools & Engineering, p.1-8. http://dx.doi.org/10.1145/1251535.1251536

Ball, T., 2008. The verified software challenge: a call for a holistic approach to reliability. *LNCS*, **4171**:42-48. http://dx.doi.org/10.1007/978-3-540-69149-5_5

Bounimova, E., Godefroid, P., Molnar, D., 2013. Billions and billions of constraints: whitebox fuzz testing in production. 35th Int. Conf. on Software Engineering, p.122-131. http://dx.doi.org/10.1109/ICSE.2013.6606558

Bovet, J., Parr, T., 2008. ANTLRWorks: an ANTLR grammar development environment. *Softw.-Pract. Exp.*, **38**(12): 1305-1332. http://dx.doi.org/10.1002/spe.872

Chen, D., Huang, R., Qu, B., *et al.*, 2014. Improving static analysis performance using rule-filtering technique. 26th Int. Conf. on Software Engineering and Knowledge Engineering, p.19-24.

Emanuelsson, P., Nilsson, U., 2008. A comparative study of industrial static analysis tools. *Electron. Notes Theor. Comput. Sci.*, **217**:5-21. http://dx.doi.org/10.1016/j.entcs.2008.06.039

Engler, D., Chen, D.Y., Hallem, S., *et al.*, 2001. Bugs as deviant behavior: a general approach to inferring errors in systems code. *ACM SIGOPS Oper. Syst. Rev.*, **35**(5):57-72. http://dx.doi.org/10.1145/502059.502041

Hajiyev, E., Verbaere, M., de Moor, O., 2006. CodeQuest: scalable source code queries with Datalog. 20th European Conf. on Object-Oriented Programming, p.2-27. http://dx.doi.org/10.1007/11785477_2

Haydar, M., Petrenko, A., Boroday, S., *et al.*, 2013. A formal approach for run-time verification of web applications using scope-extended LTL. *Inform. Softw. Technol.*, **55**(12):2191-2208. http://dx.doi.org/10.1016/j.infsof.2013.07.013

Helmick, M.T., 2007. Interface-based programming assignments and automatic grading of Java programs. 12th Annual SIGCSE Conf. on Innovation & Technology in Computer Science Education, p.63-67. http://dx.doi.org/10.1145/1269900.1268805

Hovemeyer, D., Pugh, W., 2004. Finding bugs is easy. *ACM SIGPLAN Not.*, **39**(12):92-106. http://dx.doi.org/10.1145/1052883.1052895

Hovemeyer, D., Pugh, W., 2007. Finding more null pointer bugs, but not too many. ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools & Engineering, p.9-14. http://dx.doi.org/10.1145/1251535.1251537

Jarzabek, S., 1998. Design of flexible static program analyzers with PQL. *IEEE Trans. Softw. Eng.*, **24**(3):197-215. http://dx.doi.org/10.1109/32.667879

Liu, S., Zhang, R., Wang, D., *et al.*, 2008. Implementing of Gaussian syntax-analyzer using ANTLR. Int. Conf. on Cyberworlds, p.613-618. http://dx.doi.org/10.1109/CW.2008.139

Loveland, S., 2009. Using open source tools to prevent write-only code. 6th Int. Conf. on Information Technology: New Generations, p.671-677. http://dx.doi.org/10.1109/ITNG.2009.75

Martin, M., Livshits, B., Lam, M.S., 2005. Finding application errors and security flaws using PQL: a program query language. *ACM SIGPLAN Not.*, **40**(10):365-383. http://dx.doi.org/10.1145/1103845.1094840

Panchenko, O., Treffer, A., Zeier, A., 2010. Towards query formulation and visualization of structural search results. ICSE Workshop on Search-Driven Development: Users, Infrastructure, Tools and Evaluation, p.33-36. http://dx.doi.org/10.1145/1809175.1809184

Panchenko, O., Karstens, J., Plattner, H., *et al.*, 2011. Precise and scalable querying of syntactical source code patterns using sample code snippets and a database. 19th Int. Conf. on Program Comprehension, p.41-50. http://dx.doi.org/10.1109/ICPC.2011.31

Parr, T., Fisher, K., 2011. LL(*): the foundation of the ANTLR parser generator. *ACM SIGPLAN Not.*, **46**(6):425-436. http://dx.doi.org/10.1145/1993316.1993548

Plosch, R., Gruber, H., Hentschel, A., *et al.*, 2008. On the relation between external software quality and static code analysis. 32nd Annual IEEE Software Engineering Workshop, p.169-174. http://dx.doi.org/10.1109/SEW.2008.17

Rajamani, S.K., 2006. Automatic property checking for software: past, present and future. 4th IEEE Int. Conf. on Software Engineering and Formal Methods, p.18-20. http://dx.doi.org/10.1109/SEFM.2006.10

Reinbacher, T., Brauer, J., Horauer, M., *et al.*, 2014. Runtime verification of microcontroller binary code. *Sci. Comput. Program.*, **80**(A):109-129. http://dx.doi.org/10.1016/j.scico.2012.10.015

Rutar, N., Almazan, C.B., Foster, J.S., 2004. A comparison of bug finding tools for Java. 15th Int. Symp. on Software Reliability Engineering, p.245-256. http://dx.doi.org/10.1109/ISSRE.2004.1

Schaps, G.L., 1999. Compiler construction with ANTLR and Java—tools for building tools. *Dr. Dobb's J.*, **24**(3):84-89.

Whaley, J., Avots, D., Carbin, M., *et al.*, 2005. Using Datalog with binary decision diagrams for program analysis. Asian Symp. on Programming Languages and Systems, p.97-118. http://dx.doi.org/10.1007/11575467_8

Zook, D., Pasalic, E., Sarna-Starosta, B., 2009. Typed datalog. *LCNS*, **5418**:168-182. http://dx.doi.org/10.1007/978-3-540-92995-6_12

Appendix: Part of vulnerability rules used in our experiments

This appendix presents part of vulnerability rules used in our experiments. All vulnerability rules are excerpted from PMD. The vulnerability rules are listed in Table A1. As seen, we present the names of vulnerability rules (Name), XML files that a vulnerability rule belongs to (File), and CObject expressions of vulnerability rules (CObject expression) in the table.

Table A1 Some vulnerability rules used in our experiments

Name	File	CObject expression
SimpleDateFormatNeedsLocale	design.xml	*exist*(java.text.SimpleDateFormat)
UnsynchronizedStaticDateFormatter	design.xml	*exist*(java.text.SimpleDateFormat)
RemoteSessionInterfaceNamingConvention	j2ee.xml	*exist*(javax.ejb.EJBHome)
LocalInterfaceSessionNamingConvention	j2ee.xml	*exist*(javax.ejb.EJBLocalObject)
LocalHomeNamingConvention	j2ee.xml	*exist*(javax.ejb.EJBLocalHome)
RemoteInterfaceNamingConvention	j2ee.xml	*exist*(javax.ejb.EJBObject)
ProperLogger	logging-jakarta-commons.xml	*exist*(org.apache.commons.logging.Log)
MoreThanOneLogger	logging-java.xml	*exist*(java.util.logging.Logger)
LoggerIsNotStaticFinal	logging-java.xml	*exist*(java.util.logging.Logger)
ReplaceVectorWithList	migrating.xml	*exist*(java.util.Vector)
ReplaceHashtableWithMap	migrating.xml	*exist*(java.util.Hashtable)
ReplaceEnumerationWithIterator	migrating.xml	*exist*(java.util.Enumeration)
UseArrayListInsteadOfVector	optimizations.xml	*exist*(java.util.Vector)
UseCollectionIsEmpty	design.xml	*exist*(java.util)
MDBAndSessionBeanNamingConvention	j2ee.xml	*or*(*exist*(javax.ejb.SessionBean), *exist*(javax.ejb.MessageDrivenBean))
StaticEJBFieldShouldBeFinal	j2ee.xml	*or*(*or*(*or*(*exist*(javax.ejb.SessionBean), *exist*(javax.ejb.EJBHome)), *or*(*exist*(javax.ejb.EJBLocalObject), *exist*(javax.ejb.EJBLocalHome))), *exist*(javax.ejb.EJBObject))
JUnitStaticSuite	junit.xml	*or*(*exist*(junit.framework.TestCase), *exist*(org.junit.Test))

File: the XML file that contains the XPath expressions of vulnerability rules