Rule-Based Source-Code Analysis For Detecting Security Vulnerabilities*

Hyunha Kim¹, Tae-Hyoung Choi¹, Seung-Cheol Jung¹, Oukseh Lee¹, Kyung-Goo Doh¹, and Soo-Yong Lee²

¹ Department of Computer Science and Engineering, Hanyang University, Ansan, Korea
² GTOne, Inc., Seoul, Korea

Abstract. Many security vulnerabilities related to source code have simple syntactic patterns or flow patterns that can be described as rules. In this paper, we propose a rule description language, RDL, in which we can specify simple syntactic patterns and data-flow and control-flow patterns that possibly lead to security vulnerabilities. We then introduce a universal static detector that can find the location where any patterns in source code match the given vulnerability rules in RDL. We specified a set of rules for 88 vulnerabilities from Seven Pernicious Kingdom. We implemented the detector and experimented it for the source code of Apache Tomcat 5.5.20 uncovering 47 vulnerabilities.

1 Introduction

Security vulnerabilities are abundant in modern software. According to the Gartner’s report [1], 75% of hacks occur at application level. CWE™ (Common Weakness Enumeration) [2] has been established to provide a unified, measurable set of software weaknesses, the majority of which is closely related to source code.

Hence, it is essential to get rid of those weaknesses from source code as much as possible during the phase of software development. However, most modern compilers are not usually capable of finding them. Penetration tests with a well-prepared set of test cases are preferred in practice, but hardly cover every execution path and thus are not sound in principle (false negatives).

A better alternative is the vulnerability detection by static analysis. Static analysis covers every execution path and is sound in principle (no false negatives), but might give inevitable false positives due to the approximation made by the absence of input values. Even with this shortcoming, static analysis has an edge over penetration tests because of its safety, that is, “if the detector does not find a weakness, then there will never be the weakness, guaranteed!” Hence, we choose a static approach.

* This work was supported by the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology (MEST) / Korea Science and Engineering Foundation (KOSEF), grant number R11-2008-007-02001-0.
Each of hundreds of software weaknesses currently listed in CWE has its own characteristics. Hence, in principle, there needs to be a static detector per each vulnerability. Fortunately, however, many of the vulnerabilities residing in source code share common syntactic patterns that can be mechanically identified.

In this paper, we propose a rule description language, RDL, in which we can specify simple syntactic patterns and data-flow and control-flow patterns that possibly lead to security vulnerabilities. Among the source-code-related vulnerabilities listed in CWE, many are in fact specified as rules in RDL. Given the library of vulnerable pattern rules, a universal static detector can find where any patterns in source code match the given rules. The rule-based approach has an advantage that it can fit well with the everchanging list of CWE because we only update the rules instead of developing a detector for each newly introduced vulnerability.

We implemented a rule-based security vulnerability detector. We specified in RDL a set of rules for 88 Java vulnerabilities from Seven Pernicious Kingdom (CWE-700), and experimented our detector with Apache Tomcat 5.5.20 uncovering 47 vulnerabilities.

The rest of the paper is organized as follows: Section 2 introduces the structure of our rule-based static analysis; Section 3 explains the rule description language; Section 4 presents our experimental results of detecting vulnerable spots using our vulnerability detector; Section 5 discusses related works; finally, Section 6 concludes.

2 Rule-Based Static Vulnerability Analysis

Recent years, various static-analysis techniques [3–5] have been applied to develop vulnerability-specific detectors for some well-known vulnerabilities such as buffer overflow, SQL injection, cross-site scripting (XSS), etc. Each of these vulnerabilities has its own characteristics. Thus, a distinctive, deep semantic-analysis for each vulnerability has to be employed to obtain reasonably satisfactory results – less false positives and less false negatives.

Fortunately, however, there are many vulnerabilities that do not require complicated deep-analysis to detect. Moreover, quite a few of them share the common characteristics. Hence, our approach is to design a universal program analysis for security vulnerabilities that need not require deep semantic-analysis. We develop a rule description language that can specify security vulnerabilities as syntactic-pattern and flow-pattern rules. The analysis’s main engine is a rule checker that takes the rules as inputs. With this generic analysis engine, a new detector needs not be introduced for each newly introduced vulnerability. The rule set will be updated, instead.

Our vulnerability detector consists of five parts: a rule processor, a type and property analyzer, a pattern matcher, a flow graph constructor, and a flow analyzer. The architecture of the detector is shown in Figure.1. The detector takes a program source code and a set of security rules which describe the syntactic patterns and flow patterns of software vulnerabilities.
The detector first analyzes source code to obtain types and properties, next finds program points that match the syntactic patterns provided by the given pattern rules, then extracts control flow and data flow of the source code, and finally finds program parts that have execution paths corresponding to the given flow rules.

- Rule processor: Security rules specifies either syntactic patterns or flow patterns. The rule processor takes a set of rules and transforms them into pattern specification and flow specification.
- Type and property analyzer: Before finding patterns, security vulnerability detector needs additional properties such as types, nullable values, etc. The type and property analyzer extracts type information from source code and finds properties in the code such as nullable values.
- Pattern matcher: The pattern matcher finds simple patterns described in the pattern specifications. Those are the calls of dangerous functions, the absence of necessary methods, dangerous class hierarchies, dangerous empty exception handlers, and so on.

Some vulnerabilities cannot be expressed as simple syntactic patterns. For instance, a vulnerability such as “an allocated memory cell must be disposed
before the program termination” cannot be described using a simple syntactic pattern, but can be specified using a semantic flow pattern. Then a flow analyzer that understands flow pattern rules can do the job. The program points of allocation and disposal necessary for this flow analysis are obtained by simple pattern matcher a priori.

– Flow-graph constructor: In order to find program parts that match a semantic flow pattern, we first extract the control-flow and the data-flow from source code. Both flows can be considered as graphs whose nodes are program points.
– Flow analyzer: The flow analyzer searches for the given flow patterns in the source code.

For the success of our approach, it is essential to design a novel rule description language so that each security vulnerability is precisely described in the language. The main advantage of having the rule description language is that rules and the rule checker are independent of the programming language in which the source code to be analyzed is written.

3 Rule Description Language

RDL (Rule Description Language) is a specification language to describe the syntactic and flow patterns of security vulnerabilities in source code. The abstract syntax definition of RDL in BNF is shown in Figure 2. We explain RDL through examples.

```
specification ::= {groupdef} {rule}
groupdef ::= GROUP #name = {{funcall}}
rule ::= header {constraint}
constraint ::= UNSAFE pattern {EXCLUDE pattern}
pattern ::= scopepat {arrow sort scopepat}
arrow ::= --> | <-- | => | <=
sort ::= cf | df | if
scopepat ::= [synpat] | EMPTY(block)
          | scopepat (in block | in-function id | in-subclass-of id)
block ::= CATCH(id) | FINALLY | SYNCHRONIZED | CONDITION
sympat ::= expr | decl | block
expr ::= holder | funcall | null | return | throw(id)
funcall ::= id({(holder | ...)})
decl ::= (FIELD(holder) | DEF(id)) [qualified-by id]
holder ::= id | $$ | $n | $<property>
property ::= id | NULLABLE | ARRAY | CONSTANT | UNSIGNED
```

Fig. 2. The Abstract Syntax of Rule Description Language.
3.1 Group Name Definition

A group name can be defined to stand for a set of function names. For instance, the SQL injection problem states that an dangerous input string should not be the part of a system command (CWE-089). Here, “input string” is interpreted as a result of input functions such as `System.getProperty`, `FileInputStream`, and so on. Instead of individually defining a rule for each input function, we can declare the group name of “input functions” as follows:

\[
\text{GROUP } \#\text{Ext() = \{ System.getProperty(...) FileInputStream(...) \}}
\]

Now \#Ext() denotes a set of two function names.

3.2 Rules

Each rule definition starts with the general information of each target vulnerability. For instance, the rule for CWE-586 starts with:

```
RULEID 586
RULENAME "Poor Style: Explicit call to finalize()"
ENVIRONMENT JAVA
LEVEL RISKY
```

The first line is the identification number of the vulnerability, the second is the name, the third is the platform, and the last is the danger level.

A rule definition consists of a unsafe pattern and optional excluding patterns as follows:

```
UNSAFE pattern \{EXCLUDE pattern\}
```

The vulnerability, CWE-586, states that an explicit call to `finalize()` is dangerous. However there is an exception; the call to `super.finalize()` in the body of method `finalize` is not dangerous. The rule for this vulnerability pattern is written as follows:

```
UNSAFE [finalize()]
EXCLUDE [super.finalize()] in-function finalize
```

The detailed meaning of the patterns are explained in Section 3.3–3.4.

A pattern is either a syntactic pattern or a flow pattern. The syntactic pattern can be matched just by scanning the abstract syntax tree of source code. On the other hand, the flow pattern can be matched by analyzing the flows of source code.

---

3 We use the CWE identification number for it.
3.3 Syntactic Patterns

A syntactic pattern is for capturing a code segment as is. RDL has various syntactic patterns for expressions, declarations, blocks, and scope restrictions. Note that we employ abstract constructors for expressing syntactic patterns instead of using the syntax of a specific programming language. This decision makes RDL language-independent; we can describe the security vulnerabilities of any programming languages.

Expressions It is dangerous to use functions that may invoke buffer overflow such as `strcpy` in C (CWE-242). These kinds of patterns are simple; we need to just list up the name of dangerous functions.

\[
\text{UNSAFE \{strcpy(...)\}}
\]

Variables, comparison expressions, null constants, throw statements, and return statements can also be used as patterns with placeholders. \$\$ and \$n are the placeholders for an expression, and \ldots is the placeholder for an arbitrary number of arguments.

Augmented Expressions It is dangerous to use operator \(\text{==}\) for string comparison in Java (CWE-093). The RDL rule for this vulnerability is specified as follows:

\[
\text{UNSAFE \{java.lang.String \text{== java.lang.String}\}}
\]
\[
\text{UNSAFE \{java.lang.String \text{\neq java.lang.String}\}}
\]

where \$<> is a placeholder for an expression restricted by the property specified in between \(\text{<}\) and \(\text{>}\). In the example, the placeholders restrict the arguments to have type `java.lang.String`, which means the pattern matches only when the arguments are strings. Placeholders can also restrict to be arrays, constants, signs, and nullability.

Declarations It is dangerous that a `finalize` method is declared `public` in Java (CWE-139). The RDL rule for this vulnerability is:

\[
\text{UNSAFE \{DEF(finalize())\} qualified-by PUBLIC}
\]

where \text{DEF} denotes a method (or function or procedure) declaration. Rules on variable declarations can be specified similarly.

Blocks An overly broad `catch`-block is dangerous in Java (CWE-084). Since `catch`-block is neither an expression or a declaration, RDL additionally provides the syntax for describing blocks. For instance, the constraint for CWE-084 is:

\[
\text{UNSAFE \{CATCH(Exception)\}}
\]

where \text{CATCH} denotes a `catch`-block. In addition, `finally`-blocks, `synchronized`-blocks, and condition blocks are also supported.
Scope Restriction  Some statements can be considered dangerous only when it is placed within a specific scope. In Java, a `return` statement is dangerous only when it is in a `finally`-block (CWE-087). RDL provides the syntax for restricting the scope of patterns. For instance, the RDL rule for CWE-087 is:

```
UNSAFE [return] in FINALLY
```

where “in FINALLY” restricts the scope only within `finally`-blocks. The scope can be given by the names of methods, functions, or class.

3.4 Flow Patterns

Some security vulnerabilities require more than just a syntactic scan of source code. For instance, SQL injection cannot be detected by syntactic scanning. It must be determined whether or not external input strings may become a part of an SQL command, that is, whether or not user’s inputs may flow into the arguments of SQL queries.

Our approach is to use a simple data-flow analysis for detecting SQL injection. In fact, in order to detect SQL injection precisely, we need a sophisticated program analysis which can estimate possible strings for each SQL command. Such an analysis is precise but expensive. Our rule-based approach is less precise, but inexpensive and extensible.

A flow pattern consists of syntactic patterns connected by arrows. For instance, the constraint for SQL injection is:

```
UNSAFE [#Ext()] <--if [#SQL($0)]
```

which means that it is unsafe when the results of external input functions (#Ext) may flow into the first argument of SQL query functions (#SQL). Here $0$ is a special placeholder which denotes the destination of data flow.

In order to be able to describe various flow constraints, RDL provides twelve different kinds of arrows. The formal semantics is in Figure 3 and the following is its informal explanation:

- forward and backward: the forward flow is a right arrow `-->` and the backward flow is a left arrow `<--`. For instance, $A --> B$ denotes that $B$ occurs after $A$’s occurrence. $A <-- B$ denotes that $A$ occurs before $B$’s occurrence.
- may and must: a single arrow denotes may-flow and a double arrow denotes must-flow. For instance, $A --> B$ denotes that at least one out of many flows starting from $A$ reaches $B$, and $A ==> B$ denotes that all flows starting from $A$ must reach $B$.
- control flow, and direct/indirect data flow: A flow pattern can be described based on three kinds of flows: control flow (cf), direct data flow (df), and indirect data flow (if). The control flow is considered as an execution path. For instance, $1 x=y; 2 y=z; 3 z=x$; has a control flow $1 \rightarrow 2 \rightarrow 3$. The direct data flow is a flow of values. The code has a data flow $(y \text{ at } 1) \rightarrow (x \text{ at } 1) \rightarrow (x \text{ at } 3) \rightarrow (z \text{ at } 3)$. The indirect data flow implies a possible contribution.
Assume that we have three kinds of flow digraphs: $G_{cf}$, $G_{df}$, and $G_{if}$ whose nodes are the locations of given program.

- $\text{loc}(p)$ gives a set of locations that are matched by syntactic pattern $p$.
- $\text{pathfrom}(l, G)$ gives a set of all paths $l_l l_2 \cdots l_k$ in $G$ such that $l_k$ has no out-edge.
- $\text{pathto}(l, G)$ gives a set of all paths $l_1 l_2 \cdots l_k l$ in $G$ such that $l_1$ has no in-edge.

$I$ gives a set of locations that are matched by input pattern.

$$I[p] = \text{loc}(p)$$

$$I[p\rightarrow,f] = \{ l \in I[p] : \exists l_1 \cdots l_k \in \text{pathfrom}(l, G_r). I[f] \cap \{l_1, \cdots, l_k\} \neq \emptyset \}$$

$$I[f\leftarrow,p] = \{ l \in I[p] : \exists l_1 \cdots l_k \in \text{pathto}(l, G_r). I[f] \cap \{l_1, \cdots, l_k\} \neq \emptyset \}$$

$$I[p\Rightarrow,f] = \{ l \in I[p] : \forall l_1 \cdots l_k \in \text{pathfrom}(l, G_r). I[f] \cap \{l_1, \cdots, l_k\} \neq \emptyset \}$$

$$I[f\Leftarrow,p] = \{ l \in I[p] : \forall l_1 \cdots l_k \in \text{pathto}(l, G_r). I[f] \cap \{l_1, \cdots, l_k\} \neq \emptyset \}$$

Fig. 3. The Semantics of Flow Patterns.

For instance, $x=f(y)$; has no direct data flow $y \rightarrow x$ because the value of $y$ and the return value of $f$ may be different. However, we say that the code has indirect data flow $y \rightarrow x$ because the value of $y$ may contribute the return value of $f$.

4 Experiment For Java Source Code

We implemented our rule-based detector in Objective Caml and specified a set of rules for the vulnerabilities identified in the Seven Pernicious Kingdoms. The target application of our analyzer is Apache Tomcat 5.5.20. The experiment was done in an Intel Core Duo 2.8GHz system with 2GB memory.

Table 1 shows related Java vulnerabilities from the Seven Pernicious Kingdoms. In the table, the first column is the category in the Seven Pernicious Kingdoms, the second column is the number of Java security vulnerabilities in the category, and the third column is the number of vulnerabilities covered by our tool. The fourth column lists the corresponding CWE ID numbers, and the underlined indicates the vulnerability covered by our tool.

In fact, the tool has two analysis engines: one is a RDL-based vulnerability detector presented in this paper, the other is a specialized analysis engine for specific vulnerabilities. The necessity of the latter analysis engine is from the limited expressiveness of RDL. For instance, dead code cannot be expressed in RDL. Our specialized analysis engine covers 16 vulnerabilities that comprise the problems of dead code, naming conventions, and null-pointer dereference.

The experiment result is shown in Table 2. The total of 97 security vulnerabilities were uncovered in a reasonable analysis time. Among them, 47 security vulnerabilities are uncovered by the RDL-based analysis engine.

4 The platform-specific vulnerabilities such as those in Struts and J2EE are excluded. The number of vulnerabilities covered would have been 88 if they were not excluded.
Table 1. The Target Vulnerabilities

<table>
<thead>
<tr>
<th>Category</th>
<th>Number</th>
<th>CWE Identifiers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Improper Input Validation</td>
<td>13</td>
<td>9, 15, 73, 77, 89, 99, 111, 112, 113, 114, 117, 119, 190, 470</td>
</tr>
<tr>
<td>Failure to Fulfill API Contract</td>
<td>3</td>
<td>248, 250, 252</td>
</tr>
<tr>
<td>Indicator of Poor Code Quality</td>
<td>4</td>
<td>404, 475, 476, 477</td>
</tr>
<tr>
<td>Insufficient Encapsulation</td>
<td>13</td>
<td>486, 488, 489, 491, 492, 493, 494, 495, 496, 497, 501, 582, 583</td>
</tr>
<tr>
<td>Error Handling</td>
<td>4</td>
<td>391, 395, 396, 397</td>
</tr>
<tr>
<td>Time and State</td>
<td>8</td>
<td>346, 367, 377, 378, 379, 412, 441, 472</td>
</tr>
<tr>
<td>Security Features</td>
<td>9</td>
<td>256, 258, 259, 260, 261, 272, 285, 330, 359</td>
</tr>
<tr>
<td>Environment</td>
<td>0</td>
<td>none</td>
</tr>
<tr>
<td>Total</td>
<td>54</td>
<td>28</td>
</tr>
</tbody>
</table>

5 Related Works

Dynamic detection of software vulnerabilities are popular, but does not guarantee the absence of vulnerabilities. Dynamic approaches check the vulnerabilities at run-time with test cases. These tools are relatively easy to develop and thus are abundant: CCured [6], CRED [7], Purify [8], and TinyCC [9]. A reasonable set of test cases usually reveals most of vulnerabilities. However, dynamic approaches might not be sound because in theory it is impossible to collect the ultimate test cases covering all the execution paths.

On the other hand, static approaches are able to detect all possible vulnerabilities without actually running programs. If a sound static tool concludes that there is no vulnerability in source code, there is truly none.

Most of static analysis tools are dedicated to a fixed set of software vulnerabilities. Even though Airac [10] and ASTREE [11] are precise program analysis tools, they can capture possibility of specific semantic errors such as buffer overflow, arithmetic overflow, and some other memory-related vulnerabilities. Splint [12] can find a wide range of vulnerabilities, but it is also dedicated to a fixed set. Fundamentally, those tools have no easy way to be extended for vulnerabilities that will be newly discovered. Furthermore the vulnerabilities that they detect are mainly semantic errors – many software vulnerabilities (e.g., SQL injection) are not semantic errors.

Another problem is that many static analysis tools require annotations or specifications from the users. Bandera [13] and ESC/Java [14] are precise model-checking-based static analyzers which can capture many kinds of program properties. However, in order to using them, the users should put annotations into the code. This requirement is not desirable for a practical tool to detect software vulnerability.

In comparison with the tools addressed, our tool has advantages. Our tool is sound; it can guarantee absence of vulnerabilities. Our tool is flexible; it can be
Table 2. The Experimental Result

<table>
<thead>
<tr>
<th>filename</th>
<th>lines</th>
<th>analysis time(s)</th>
<th># detection (RDL only)</th>
<th># detection (+specialized)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AccessLogValve</td>
<td>1144</td>
<td>0.860</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ApplicationDispatcher</td>
<td>1029</td>
<td>0.897</td>
<td>4</td>
<td>13</td>
</tr>
<tr>
<td>ASCIIReader</td>
<td>204</td>
<td>0.795</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CGI Servlet</td>
<td>1962</td>
<td>1.150</td>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>ClassLoaderFactory</td>
<td>260</td>
<td>1.296</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ContextConfig</td>
<td>1359</td>
<td>1.038</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>DeleteContextAction</td>
<td>161</td>
<td>0.698</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>DeleteHostsAction</td>
<td>155</td>
<td>0.703</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>DeleteRealmsAction</td>
<td>148</td>
<td>0.681</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>EditContextAction</td>
<td>239</td>
<td>1.005</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>FarmWarDeployer</td>
<td>741</td>
<td>0.885</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>FileMessageFactory</td>
<td>312</td>
<td>0.751</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>JDBCAccessLogValve</td>
<td>690</td>
<td>0.773</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>JkMX</td>
<td>395</td>
<td>1.265</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>JMXAccessorTask</td>
<td>732</td>
<td>1.320</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>JspC</td>
<td>1406</td>
<td>1.294</td>
<td>6</td>
<td>23</td>
</tr>
<tr>
<td>JspConfig</td>
<td>472</td>
<td>1.602</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>JvmRouteBinderValve</td>
<td>546</td>
<td>1.043</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>MemoryRealm</td>
<td>329</td>
<td>1.226</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PersistentManagerBase</td>
<td>1130</td>
<td>1.347</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>ProxyDirContext</td>
<td>1621</td>
<td>1.339</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>RealmBase</td>
<td>1436</td>
<td>1.619</td>
<td>4</td>
<td>11</td>
</tr>
<tr>
<td>ReplicationValve</td>
<td>659</td>
<td>1.303</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SetNextRule</td>
<td>215</td>
<td>1.181</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SetRootRule</td>
<td>216</td>
<td>1.173</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SetTopRule</td>
<td>216</td>
<td>1.189</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>StandardWrapperValve</td>
<td>386</td>
<td>1.301</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>total</strong></td>
<td></td>
<td><strong>47</strong></td>
<td><strong>97</strong></td>
<td></td>
</tr>
</tbody>
</table>

extended for vulnerabilities discovered in the future. Our tool is fully automatic; it does not require any annotation inside an input program.

There are some works that have similar advantages to ours. UNO [15] is a model-checking tool which statically analyzes an input program with a vulnerability description written in a C-like language. Flow patterns as well as syntactic patterns can be described in this language. However, since the goal of UNO is to analyze only C programs, the description language cannot express complex syntactic patterns that are necessary for Java. Our RDL is designed for multiple programming languages such as C and Java so that it is not difficult to extend it for other programming languages. A commercial tool, Fortify [16], seems to have a similar mechanism to ours. However, it is not easy to compare ours with Fortify because the details of its engine is not open.
6 Conclusion

We design a specification language RDL to describe simple syntactic patterns as well as data-flow and control-flow patterns. Most of the security vulnerabilities listed in CWE can be specified in RDL. Moreover, RDL is designed so that new rules can be easily added.

However, there still remain vulnerabilities that cannot be expressed in RDL. For instance, “B should be on the flow from A to C.” In order to remedy this deficiency, we should employ more complicated flow analysis. It may be the trade-off between that we expand RDL to accommodate more complicated rules and that we develop a separate detector for some vulnerabilities that require more sophisticated flow analysis.

References

1. Lanowitz, T.: Now is the time for security at the application level. Gartner Report G00127407, Gartner (December 2005)
3. Fasoo.com, Inc: SPARROW: Catching software bugs early at build time (July 2007)
