NEGWeb: Static Defect Detection via Searching Billions of Lines of Open Source Code *

Suresh Thummalapenta
Department of Computer Science
North Carolina State University
Raleigh, USA
sthumma@ncsu.edu

Tao Xie
Department of Computer Science
North Carolina State University
Raleigh, USA
xie@csc.ncsu.edu

ABSTRACT
To find defects in programs, existing approaches mine programming rules as common patterns out of program source code and classify defects as violations of these mined programming rules. However, these existing approaches often cannot surface out many programming rules as common patterns because these approaches mine patterns from only one or a few project code bases. To better support static bug finding based on mining code, we develop a novel framework, called NEGWeb, for substantially expanding the mining scope to billions of lines of open source code based on a code search engine. NEGWeb detects violations related to neglected conditions around individual API calls. We evaluated NEGWeb to detect violations in local code bases or open source code bases. In our evaluation, we show that NEGWeb finds three real defects in Java code reported in the literature and also finds three previously unknown defects in a large-scale open source project called Columba (91,508 lines of Java code) that reuses 2225 APIs. We also report a high percentage of real rules among the top 25 reported patterns mined for five popular open source applications.

1. INTRODUCTION
To improve software quality, developers can write down programming rules and then apply static or runtime verification tools to detect software defects related to violations of these rules. But in practice, these programming rules often do not exist or even when they exist, they are written informally, not amenable to verification tools. To tackle the issue of lacking programming rules, various approaches have been developed in recent years to mine programming rules from program executions [3, 6, 18], program source code [1, 2, 4, 5, 10, 12, 13, 15], or version histories [11, 16]. Then these approaches detect likely defects as those violations of the mined programming rules. A common methodology adopted by these approaches is to discover common patterns (e.g., frequent occurrences of pairs or sequences of API calls) across a sufficiently large number of data points (e.g., code locations). Then these common patterns often reflect programming rules that should be obeyed when programmers write code using a similar set of rule elements such as API calls and condition checks. However, these existing approaches often cannot surface out many programming rules as common patterns because these approaches mine patterns from a small number of project code bases and there are often too few data points in these code bases to support the mining of desirable patterns. In other words, the number of data points to support a pattern related to a particular programming rule is often insufficient. This phenomenon is reflected on the empirical results reported by these exiting approaches: often a relatively small number of real programming rules mined from huge code bases.

A natural question to ask is how we can address this issue of lack of relevant data points in mining programming rules. Code Search Engines (CSEs) such as Google code search [7] and Koders [9] give us some hope. These CSEs can be used to assist programmers by providing relevant code examples with usages of the given query from a huge number of publicly accessible source code repositories.

To address the issue of lacking relevant data points in mining programming rules, we develop a novel framework, called NEGWeb, for static bug finding based on searching and mining billions of lines of open source code with the help of a code search engine such as Google code search [7]. Our new approach is the first bug finding approach with this scale and based on a Code Search Engine (CSE). Our previous work also developed the MAPO [17] and PARSEWeb [14] approaches for mining source files returned by a CSE but these previous approaches focus on mining API usage patterns to assist programmers to write API client code effectively. In contrast, our new NEGWeb framework focuses on static bug finding based on mining programming rules, which poses a different set of mining requirements. In static bug finding based on mining, one important challenge is to reduce false positives (e.g., reported warnings that do not indicate real defects or patterns that do not reflect real programming rules).

To address the issue of false positives in static bug finding, we develop NEGWeb to detect violations related to neglected conditions around individual API calls, in particular, (1) missing conditions that check the receiver or arguments of an API call before the API call or (2) missing conditions that check the return values or receiver of an API call after the API call. As shown by Chang et al. [4] in their
approach of revealing neglected conditions, neglected conditions are quite common among the defects in programs. Different from their approach, which focuses on mining one project code base, NEGW eb mines a much larger scope of code bases based on a CSE. In addition, NEGW eb’s mining based on simple statistical analysis is more scalable than their approach based on frequent sub-graph mining with known scalability issues.

While enjoying the benefits provided by a CSE in terms of expanding the analysis scope, NEGW eb faces one challenge that existing approaches do not face: the code samples returned by a CSE are often partial and not compilable, because the CSE retrieves individual source files with usages of the given query method, instead of entire projects. We develop several new heuristics along with previously developed heuristics [14] to tackle this challenge.

This paper makes the following main contributions:
- A novel framework for static bug finding based on a CSE. Our framework is the first bug finding approach that can deal with that large scale of open source code through a CSE.
- A set of rule templates for describing common neglected conditions around individual API calls. These templates allow us to exploit program dependencies among rule elements (e.g., condition checks and API calls) to reduce false positives.
- A technique for analyzing partial code samples through Abstract Syntax Trees (AST) and Directed Acyclic Graphs (DAG) and a set of heuristics for further reducing false positives.
- An Eclipse plugin implemented for the proposed framework and several evaluations to assess the effectiveness of the tool. In particular, NEGW eb confirms three real defects in Java code reported in the literature and also detects three previously unknown defects in a large-scale open source project that reuse 2225 APIs. We also report a high percentage of real rules among the top 25 reported patterns mined for five open source applications.

The rest of the paper is organized as follows. Section 2 explains the framework through an example. Section 3 describes key aspects of the framework. Section 4 discusses evaluation results. Section 5 discusses the threats to validity. Section 6 presents related work. Finally, Section 7 concludes.

2. EXAMPLE

We next use an example to describe how our NEGW eb framework mines condition patterns from billions of lines of open source code available on the web and uses the mined condition patterns to detect violations in either an input source application or in available open source projects on the web. We use the class org.apache.bcel.verifier.Verifier and its methods doPass1, doPass2, doPass3a, and doPass3b from the BCEL library as an illustrative example for explaining our framework. The Verifier class is mainly used for verifying generated class files.

Given an API such as Verifier and its methods, NEGW eb constructs a query with the class name and gathers relevant code samples from a CSE. An example code sample gathered from a CSE is shown in Figure 1. NEGW eb parses each code sample and transforms the sample into an intermediate form, represented in the form of a Directed Acyclic Graph (DAG). NEGW eb uses dominance and data-dependency concepts, and gathers preceding and succeeding conditions around the nodes that include any of the methods such as doPass1 or doPass2. NEGW eb identifies different condition patterns for RECEIVER, ARGUMENT, and RETURN objects around the given method. A few condition patterns identified from the example code sample are shown as below.

```
01: doPass1 RECEIVER NULLITY
02: doPass2 PRE_METHOD CONST_EQUAL doPass1
03: doPass3 RETURN CONST_EQUAL VerificationResult.VR_OK
04: doPass3a ARGUMENT GEN_EQUALITY
```

Each condition pattern consists of the method name, pattern type, condition type, and optional additional information separated by spaces. The condition pattern in Line 1 describes that before invoking the doPass1 method, a NULLITY check must be done on the receiver variable of that method. An example of the pattern type PRE_METHOD shown in Line 2 describes that the method doPass2 must be invoked only after the doPass1 method. Line 3 describes that the return value of the doPass1 method should be compared with the constant VerificationResult.VR_OK. Line 4 describes the ARGUMENT pattern type where the argument must be verified before invoking the doPass3a method.

NEGW eb mines the extracted condition patterns to compute frequent condition patterns, referred as mined patterns. NEGW eb applies these mined patterns on either the input source application or gathered code samples to detect violations. For example, consider the code sample below taken from existing open source projects with violations:

```
Verifier v = VerifierFactory.getVerifier(args[k]);
VerificationResult vr;
v = v.doPass1();
v = v.doPass2();
```

NEGW eb detects a violation from the preceding code sample as the sample violated the condition pattern “doPass1 RETURN CONST_EQUAL VerificationResult.VR_OK”. Sometimes, the same violation can appear multiple times because the code sample can violate multiple condition patterns. For example, the preceding code sample also violates the condition pattern “doPass2 PRE_METHOD CONST_EQUAL doPass1”.

3. FRAMEWORK

Our NEGW eb framework consists of seven major components: the application scanner, code search engine, code downloader, code analyzer, pattern extractor, pattern miner, and anomaly detector. Figure 2 shows an overview of all
Our framework uses Google code search (GCSE) [7] for collecting related samples because of two main reasons: (1) GCSE provides client libraries that can be used by other tools to interact with and (2) GCSE has public forums that provide good support. However, our framework is independent of GCSE and can leverage any other CSE to gather related samples.

3.3 Code Downloader

The code downloader accepts the $IC_x$ set and their associated $IM_{xy}$ sets as input, and interacts with CSE for searching and gathering related samples. To improve performance, the code downloader constructs only one query for each $IC_x$ instead of different queries for each pair $(IC_x, IM_{xy})$. The gathered code samples are applicable to all $IM_{xy}$ of the $IC_x$ that is included in the query. An example code sample gathered from CSE for the query “org.apache.bcel.verifier.Verifier” related to the BCEL library is shown in Figure 1. The gathered code samples are stored in a repository and is referred as a local source code repository.

3.4 Code Analyzer

The code analyzer accepts the local source code repository as input and analyzes the code samples statically through Abstract Syntax Trees (AST) to construct DAGs. The code analyzer uses several type heuristics while analyzing these code samples as the code samples gathered through CSE are partial. The reason for the partial nature of these code samples is that the CSE extracts only the source files with usages of the given query instead of entire projects. These type heuristics help identify the object types in the code samples. For example, for the method $doPass1$ in Statement 6, the type heuristics identify the return type as $VerificationResult$ from the left-hand side of the assignment statement.

The constructed DAG consists of two kinds of nodes: control and non-control. Control nodes, referred as $CT$, represent the control-flow statements such as if, while, and for, which control the flow of the program execution. Non-control nodes, referred as $NT$, represent statements such as method-invocations or type casts. For example, Statement 5 in the code sample is a control node and Statement 6 is a non-control node. While encountering a control node, say $CT_i$, the code analyzer also identifies all variables, say $\{V_1, V_2, ..., V_n\}$, that participate in the conditional expression of that node. Each variable $V_i$ is associated with its corresponding condition type $T_i$. Therefore, a control node $CT_i$ includes a set of pairs $\{(V_1, T_1), (V_2, T_2), ..., (V_n, T_n)\}$. For example, the control node $CT_5$ (suffix indicates the statement id) includes $\{(\text{verf, NULLITY})\}$ pair. The possible values for $T_i$ are shown in Table 1. The table also shows the description and additional information associated with each $T_i$. The additional information represents operators, constants, or method-invocations associated with each pair $(V_i, T_i)$. For example, the control node $CT_{10}$ includes the pair $\{(\text{vr1, CONST_EQUAL})\}$ and the additional information associated with this pair includes the constant $\text{VerificationResult.VR_OK}$.

In general, the variable $V_i$ can participate in the conditional expression either directly or indirectly. A direct participation refers to the scenario where the conditional check is conducted directly on the $V_i$ such as the NULLITY condition type shown in Table 1. An indirect participation refers to the

3.1 Application Scanner

The application scanner accepts a source application as input and gathers the external classes and methods used by that application. A class is recognized as an external class if the package name of that class does not belong to the set of package names of the given application. The application scanner initially collects all packages and classes of the source application, and then uses the collected information to identify the external classes through their package names. The application scanner also gathers the set of methods referred by the source application for each such external class. The set of external classes and methods is provided as input to the code downloader.

3.2 Code Search Engine

Code Search Engines (CSE) such as Google [7] and Koders [9] are primarily used by programmers in searching for related samples from the available open source projects on the web. As CSEs can search billions of lines of source code available on the web, CSEs can serve as powerful resources of open source code. Due to the strength of these CSEs in searching for related samples, these CSEs can be exploited for other tasks such as detecting violations in the applications that reuse existing open source projects. Therefore, our framework uses CSEs to gather related samples for the given set of APIs.
### Table 1: Values of condition types, $T_i$, associated with variable, $V_j$, in a conditional expression.

<table>
<thead>
<tr>
<th>Condition Type ($T_i$)</th>
<th>Description</th>
<th>Additional Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULLITY</td>
<td>direct null check. Eg: if($V_j == null$) { ... }</td>
<td>operator involved, say $=!$</td>
</tr>
<tr>
<td>M_NULLITY</td>
<td>indirect null check. Eg: if($M_k(V_j) == null$) { ... }</td>
<td>operator involved, say $==$</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>if the variable type is boolean. Eg: if($V_j$) { ... }</td>
<td>method-invocation, say $M_k$</td>
</tr>
<tr>
<td>M_BOOLEAN</td>
<td>indirect boolean check. Eg: if($M_k(V_j)$) { ... }</td>
<td>method-invocation, say $M_k$</td>
</tr>
<tr>
<td>CONST_EQUAL</td>
<td>if the variable is compared with a constant. Eg: if($V_j == SUCCESS$)</td>
<td>operator involved, say $===$</td>
</tr>
<tr>
<td>M_CONST_EQUAL</td>
<td>indirect constant equality check. Eg: if($M_k(V_j) == SUCCESS$)</td>
<td>constant value, say SUCCESS</td>
</tr>
<tr>
<td>RETVAL_EQUAL</td>
<td>if the variable is compared with the return value of a method-invocation. Eg: if($V_j &lt; M_l$)</td>
<td>method-invocation, say $M_l$</td>
</tr>
<tr>
<td>M_RETVAL_EQUAL</td>
<td>if the variable is compared indirectly with the return value of a method-invocation. Eg: if($M_k(V_j) &gt; M_l$)</td>
<td>method-invocation, say $M_l$</td>
</tr>
<tr>
<td>INSTANCE_CHECK</td>
<td>if the conditional check involves <code>instanceof</code> operator. Eg: if($V_j instanceof Integer$)</td>
<td>type-name, say Integer</td>
</tr>
<tr>
<td>GEN_EQUALITY</td>
<td>if the conditional check does not fall in preceding types. Eg: if($V_j &lt; V_a$) { ... }</td>
<td>other expression, say $V_a$</td>
</tr>
</tbody>
</table>

scenario where $V_j$ is an argument of a method-invocation, say $M_k$, that is involved in the conditional expression. For example, the condition type R_NULLITY refers to the preceding scenario. To distinguish the direct and indirect participations, the condition types of indirect participation are prefixed with “M_”.

The constant values shown in Table 1 can be literals or constant variables. The code analyzer identifies constant variables such as SUCCESS or FAILURE by using a heuristic based on general guidelines of the Java programming language. The guidelines describe that variable names with all capital letters can be treated as constants.

While constructing the DAG, the code analyzer identifies nodes in the graph that contain $IM_{xy}$ and marks those nodes as APINodes, referred as $AN_{i}$. The constructed DAG can contain one or more $AN_{i}$ nodes and this DAG serves as a Control Flow Graph (CFG) for the pattern extractor that identifies the condition patterns around each $IM_{xy}$. In the example code sample, the code analyzer identifies Statements 6, 9, 13, and 14 as APINodes.

### 3.5 Pattern Extractor

The pattern extractor accepts the constructed DAG as input from the code analyzer and performs forward and backward traversal over each $AN_{i}$ node to identify condition patterns around each $IM_{xy}$. In particular, the pattern extractor uses the concept of dominance with a blend of control-flow and data-flow dependencies. The condition patterns identified by the pattern extractor can be classified into two major categories: preceding and succeeding patterns. We next describe how the pattern extractor identifies these preceding and succeeding condition patterns.

#### 3.5.1 Preceding Condition Patterns

The pattern extractor extracts preceding condition patterns by using the concept of dominance. The definition of dominance concept is given below:

**Dominance:** A node $N$ dominates another node $M$ in a control flow graph (represented as $N$ dom $M$) if every path from the starting node of the CFG to $M$ includes $N$. Initially, the pattern extractor identifies the dominant $CT_i$ nodes for each $AN_{i}$ node. For example, the pattern extractor identifies that $CT_5$ dominates $AP_5$. The pattern extractor computes an intersection between the variable set associated with the $CT_i$ node, say $\{V_1, V_2, ..., V_n\}$, and the receiver or argument variables of the $AN_i$ node, say $\{REC_i, ARG_{k1}, ..., ARG_{k2}\}$. If the intersection $\{V_1, V_2, ..., V_n\} \cap \{REC_i, ARG_{k1}, ..., ARG_{k2}\} \neq \Phi$, then the pattern extractor checks whether the $AN_i$ node is data-dependent on the $CT_i$ node. The data-dependency check ensures that the variable involved in the $CT_i$ node is not re-defined in the path between $CT_i$ and $AN_i$ nodes. If the $AN_i$ node is data-dependent on the $CT_i$ node, the pattern extractor extracts the associated condition pattern. The associated condition pattern for nodes $CT_i$ and $AN_i$ in the example code sample is “doPass1 REVEIVER NULLITY”, which indicates that a NULLITY check must be done on the the receiver variable of the method doPass1. The general condition pattern format extracted by the pattern extractor includes $IM_{xy}$, pattern type, and condition type ($T_i$). However, for some condition patterns, a fourth element that represents other method-invocations is included in the pattern format.

The pattern extractor extracts three types of preceding condition patterns: RECEIVER, ARGUMENT, and PRE_METHOD. We next describe the details of each pattern type.

**RECEIVER:** represents condition patterns on the receiver variable preceding the call site of $IM_{xy}$. For example, the receiver condition pattern associated with $CT_5$ and $AN_6$ for the method doPass1 is “doPass1 RECEIVER NULLITY”. **ARGUMENT:** represents condition patterns on the argument variable preceding the call site of $IM_{xy}$. For example, the argument condition pattern associated with $CT_{12}$ and $AN_{13}$ for the method doPass3a is “doPass3a ARGUMENT GEN_EQUALITY”.

**PRE_METHOD:** represents condition patterns on other method-invocations of the same receiver variable preceding the call site of $IM_{xy}$. For example, the PRE_METHOD condition pattern associated with $CT_7$ and $AN_9$ for the method doPass2 is “doPass2 PRE_METHOD CONST_EQUAL doPass1”. This pattern indicates that before invoking the doPass2 method, the method doPass1 must be invoked and a condition check must be performed on the return value of the method doPass1.

#### 3.5.2 Succeeding Condition Patterns

The pattern extractor extracts the succeeding condition patterns by using the concept of post-dominance. The pat-
Anamoly Detector

The anamoly detector operates in two modes. In Mode 1, the anamoly detector uses mined heuristics to detect violations in the source application. In Mode 2, the anamoly detector uses mined heuristics to detect violations in the gathered code samples. In particular, the anamoly detector re-extracts the condition patterns for each $IM_{xy}$, and checks whether the newly extracted condition patterns contain the mined patterns. Any missing mined patterns are reported as violations. For each detected violation, the anamoly detector assigns a confidence level, which is the same as the confidence level of the associated condition pattern.

Additionally, the anamoly detector uses two heuristics to reduce the number of false positives and to sort the detected violations based on their importance.

**Anamoly Heuristic 1:** A violation detected for succeeding condition patterns can be ignored if the corresponding variable is a part of the return statement of the enclosing method declaration or an argument of another method-invocation.

This heuristic is based on our experience with different subjects where an expected conditional check often appears in the call site of $IM_{xy}$. The rationale behind this assumption is that a group of condition patterns can often appear together in code samples. For example, consider that two RETURN condition patterns, say SUCCESS and FAILURE, appeared together in 10 code samples. The computed support values for these two condition patterns will be 0.5 each, resulting in a low confidence level. However, these condition patterns appeared in all code samples and should have a high confidence level. We use the value of $ECP_{xy}$ to classify non-high condition patterns into confidence levels AVERAGE and LOW. We used values 0.75 for UT and 0.1 for LT. These values are based on our empirical experience with different subjects.

**Anamoly Heuristic 2:** A violation detected for a call site with no conditional checks around can be given higher preference than violations detected for other call sites with a few conditional checks around.

The rationale behind this heuristic is that call sites with no conditional checks can have a higher chance of being a defect than call sites with a few conditional checks around. The anamoly detector sorts the detected violations based on the attributes confidence level, support of the related condition pattern, and the favorable number of code samples.
4. EVALUATION

We conducted four different evaluations on NEGWeb to show that NEGWeb can effectively mine real rules from related code samples gathered through a CSE, and can be effective in identifying real defects. In the first evaluation, we used two applications and three API libraries to mine condition patterns and manually confirmed the extracted top 25 condition patterns through the available documentation and source code of the applications. For generality, we refer all subjects as applications. In the second evaluation, we applied the mined condition patterns in a novel way to detect violations in available open source applications. As NEGWeb mainly targets at neglected conditions that help increase the robustness of applications, we manually confirmed the top 50 violations as defects or other categories through inspection. In the third evaluation, we verified whether NEGWeb can confirm known Java defects reported in the literature by earlier related approaches. In the fourth evaluation, we conducted a case study with a large-scale application called Columbia. The details of subjects and results of our evaluation are available at http://ase.csc.ncsu.edu/negweb/

4.1 Open Source Applications

In this section, we describe condition patterns mined by NEGWeb for two open source applications and three API libraries that vary in size and purpose. We also apply the mined condition patterns onto the gathered code samples to detect violations in available open source projects.

4.1.1 Condition Patterns

The characteristics such as the number of classes and methods of the five applications used for mining condition patterns are shown in Columns “Classes” and “Methods” of Table 2. The Java Util package includes the collections framework and other popular utilities used by many different applications. The BCEL library, developed by Apache, is mainly used to analyze, create, and manipulate Java class files. The Hibernate framework abstracts relational databases into an object-oriented methodology. Java servlets and Java Transactions are industry standards for developing multi-tier server-side Java applications. The common reason for selecting these applications is the presence of condition patterns as described by their associated documentations that can help confirm the real patterns. We feed the set of classes and methods of each application as input to NEGWeb.

Column “Samples” of Table 2 shows the number of code samples gathered and analyzed for each application from the code search engine. For example, NEGWeb gathered and analyzed 49,858 code samples for Java Util packages. The number of condition patterns mined for each application is shown in Column “Patterns” of Table 2. We manually analyzed the first 25 patterns of each application and classified them into three categories: rules, usage patterns, and false positives. Rules describe the properties that must be satisfied for using an API, whereas usage patterns are common ways of using an API. We used the available on-line documentations, JML specifications, or the source code of the application for classifying the mined condition patterns into these three categories. As shown in Table 2, most of the mined patterns are classified as rules and a few are classified as false positives. The number of false positives is a little more for Java Servlet, because of one common pattern that appeared among these false positives. The common pattern is “ServletRequest INSTANCE_CHECK HttpServletRequest”, which describes that an instance check has to be performed with the ServletRequest class before invoking its methods. Although this pattern is a common usage, the available specification of Java Servlet does not confirm this pattern as a real rule. The primary reason for the lesser number of false positives in other subjects is due to the large number of analyzed data points gathered through CSE.

We manually classified all condition patterns of the Java Util package. The primary reason for selecting the Java Util package for manual analysis is the availability of JML specification that can help confirm the mined patterns. The classified categories of all patterns for the Java Util APIs are shown in the last row of Table 3. Among all mined patterns, the real rules constitute 56.25% (36/64) and usage patterns constitute 26.56% (17/64). A non-negligible percentage of 17.18% (11/64) is classified as false positives. However, the number of false positives in the top 25 rules shown in Table 2 is zero. This evaluation shows the effectiveness of our mining heuristics that surface out real rules by ranking false positives below. Table 3 further shows the classification of the mined condition patterns based on the pattern types of NEGWeb. NEGWeb is effective in extracting and mining real rules for pattern types PRE_METHOD and RETURN, which are usually the main sources of neglected conditions. The pattern type SUCC_METHOD has the largest number of false positives.

We next describe the mined patterns for the Matcher class of Java Util packages. NEGWeb identified 10 patterns for this class, which is an engine that performs matching operations on a character sequence by interpreting a given regular expression. For each mined pattern, we show the method

Table 2: Condition patterns mined by NEGWeb and their violations.

<table>
<thead>
<tr>
<th>Application</th>
<th>Input Application</th>
<th>CSE</th>
<th>Categories of first 25 patterns</th>
<th>Time (in min.)</th>
<th># Violations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java Util APIs</td>
<td>19</td>
<td>144</td>
<td>49858</td>
<td>64</td>
<td>20</td>
</tr>
<tr>
<td>BCEL</td>
<td>357</td>
<td>2691</td>
<td>9697</td>
<td>322</td>
<td>20</td>
</tr>
<tr>
<td>Hibernate</td>
<td>1253</td>
<td>11452</td>
<td>32486</td>
<td>542</td>
<td>21</td>
</tr>
<tr>
<td>Java Servlet APIs</td>
<td>19</td>
<td>89</td>
<td>16628</td>
<td>54</td>
<td>18</td>
</tr>
<tr>
<td>Java Transaction APIs</td>
<td>7</td>
<td>37</td>
<td>5555</td>
<td>15</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 3: Classification of mined patterns of Java Util package.

<table>
<thead>
<tr>
<th>Pattern type</th>
<th>#Total</th>
<th>#Rule</th>
<th>#UP</th>
<th>#FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>RECEIVER</td>
<td>5</td>
<td>3</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>ARGUMENT</td>
<td>6</td>
<td>5</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>PRE_METHOD</td>
<td>12</td>
<td>11</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>RETURN</td>
<td>19</td>
<td>12</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>SUCC_METHOD</td>
<td>22</td>
<td>5</td>
<td>10</td>
<td>7</td>
</tr>
<tr>
<td>SUCC_RECEIVER</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SUM</td>
<td>64</td>
<td>36</td>
<td>17</td>
<td>11</td>
</tr>
</tbody>
</table>

UP: usage pattern, FP: false positive
name, pattern type, condition type, and additional information. The additional information is optional and is shown for pattern types such as PRE_METHOD and SUC_METHOD.

find, RETURN, BOOLEAN,
start, PRE_METHOD, BOOLEAN, find
detect violations. The total number of violations for each API is shown in Column “Total”. The manual classification results of the first 50 violations are shown in Columns “Defect”, “CS”, “WP”, “Hint”, and “FP”. Except for BCEL, the number of false positives is quite low for APIs of other applications. The reason for a high number of false positives in BCEL is due to limitations in the current NEGWeb implementation such as not handling conditional expressions in assignment statements, which we plan to address in near future work without difficulty.

We next describe details of defects detected in open source applications for Java Servlet APIs. The API used from this application is “ServletConfig, getConfig(String)” and the condition pattern used is “getInitParameter RETURN NULLITY”, which indicates that a nullity check must be performed on the return value of the getInitParameter method. We confirmed this pattern from the JTA specification, which describes that this method can return null, if the parameter does not exist. However, 31 open source projects violated this mined condition pattern. Among the top 50 violations, 38 violations are classified as defects in our inspection. We found some more interesting facts during this evaluation. We use the below code sample that is collected from the existing open source projects to describe these facts.

```java
...String jspCP = config.getInitParameter("jspCP");
if (jspCP != null) {
  ...
  this.javaEncoding = config.getInitParameter("javaEncoding");
}
```

In the preceding code sample, the getInitParameter method is used twice. However, the NULLITY check on the return value is done only once, and is ignored during the second invocation. As the getInitParameter method can return null, the absence of NULLITY check can cause NullPointerException. We also found that the same piece of code that has violations is used in different applications. For example, we found a similar violated code in open source projects tomcat, fisheye, and jboss for the getInitParameter method. As programmers often tend to copy related code from existing applications, the violations can also propagate from applications to applications. Our results show the number of neglected conditions that exist in the available open source applications and the necessity for an approach such as NEGWeb.

### 4.2 Real defects from the literature

We evaluated NEGWeb to check whether it can detect known defects described in the literature. We picked two
defects in the AspectJ application detected by JADET\(^2\) [15] and two defects in Java SSE Library and Joelq detected by DIDUCE [8]. We next describe details of these defects and explain the evaluation results with NEGWeb.

### 4.2.1 Defects Detected by JADET

In the AspectJ application, JADET detected two defects related to loops that are incorrectly executed at most execution. We show the code sample taken from JADET as below:

```java
private boolean verifyNIAP(...) {
    Iterator iter = ...;
    while(iter.hasNext()) {
        ... = iter.next(); ...;
        return verifyNIAP(...);
    }
}
```

As shown in the preceding code sample, the `return` statement in the `while` loop causes the method to return without iterating all elements in the `Iterator`. NEGWeb confirmed this defect with a support value of 0.708. The second defect in the Joelq application is related to not checking the return value of the `read` method of the class `InputStream`. The method `read` returns the number of bytes that are actually read; programmers often forget to check whether the number of read bytes is equal to the expected number of bytes to be read. NEGWeb confirmed this defect with a support value of 0.708.

### 4.2.2 Defects Detected by DIDUCE

We collected two defects reported by DIDUCE in applications Java SSE and Joelq. The defect in the Java SSE library is related to not handling the return value of the `read` method of the class `InputStream`. The method `read` returns the number of bytes that are actually read; programmers often forget to check whether the number of read bytes is equal to the expected number of bytes to be read. NEGWeb confirmed this defect with a support value of 0.708.

The second defect in the Joelq application is related to not checking the return value of the method `put` of the class `Hashtable`. When an object is inserted into the `Hashtable` through the `put` method, the method either returns an existing associated object with that key value or returns `null`. NEGWeb could not confirm this defect as the support for the extracted pattern is low. Among 77 related code samples gathered from the code search engine, NEGWeb detected that only 5 code samples have the `NULLITY` check on their return value. However, this defect mainly depends on the semantics of the logic applied to the extraction of the commonality of the API. Therefore, reporting violations based on these kinds of patterns can result in a large number of false positives. We want to emphasize that the motivation of NEGWeb is mainly to mine the most common condition patterns that can cause potential defects and to reduce the number of false positives among the detected violations.

\(^{2}\)JADET reported three defects in the paper. One defect with BCEL APIs is not related to neglected conditions and does not fall into the scope of our current approach.

### 4.3 Case Study: Columba

Columba 1.4\(^3\) is an open source email client application written in Java. Columba provides a user-friendly graphical interface and is suitable for internationalization support. The Columba application includes 1165 classes and 6894 methods that contain a total of 91,508 lines of Java code. We used NEGWeb to mine rules and detect violations in the Columba application. NEGWeb identified the external classes and methods used by Columba and mined condition patterns for those classes by interacting with APIs. NEGWeb gathered and analyzed 309,757 code samples for mining these condition patterns. NEGWeb applied the mined condition patterns to detect violations in Columbia. NEGWeb mined 559 condition patterns related to the external classes and methods used by Columba. Among these 559 condition patterns, 370 condition patterns were used to detect 1647 violations. Each mined condition pattern can be used to detect multiple violations at different locations of source code. NEGWeb manually analyzed violations of the first 25 mined patterns. As each mined pattern can detect multiple violations at different locations of source code, these 25 patterns detected 70 violations. We classified these 70 violations into different violation categories using the same classification criteria described in Section 4.1.2.

The results of our evaluation are shown in Table 5. Each row in the table represents a mined pattern. Columns “Supp.”

---

\(^{3}\)http://sourceforge.net/projects/columba/
and “PC” give the support and manually assigned category of each pattern, respectively. Column “Total” gives the total number of violations detected by that pattern. Among the first 25 mined patterns, 19 patterns are classified as real rules, 2 are classified as usage patterns, and 4 are classified as false positives. The results also show that the top 10 patterns do not have any false positives. As each pattern type can be used to detect multiple violations of different categories, depending on the usage in the source code, we show how many of the total violations of each pattern fall into different violation categories. In total, there are 70 violations among which, 3 are defects, 7 are code smells, 9 are wrappers, 37 are hints, and 14 are false positives. All three defects among 70 violations are detected with the top 10 mined patterns. In general, a false-positive pattern leads to a false-positive violation. For example, Patterns 11 and 14 are false positives that caused 5 violations of the false-positive category. However, the additional 3 false positives highlighted in the table are due to limitations in the current NEGW eb implementation such as not handling conditional expressions in assignment statements. We plan to address these limitations in near future work and these limitations can be addressed without any difficulty. The largest number of hints are detected with Pattern 4, which is “next SUC_METHOD BOOLEAN hasNext” of the Iterator class. Although this mined pattern is a real rule, lists are used for storing a single object instead of multiple objects in Columba. Therefore, we classified these violations as hints that can help improve the maintainability and readability of code.

We next describe defects detected by NEGW eb in Columba. We confirmed these defects through inspecting the source code of the associated class and call sites of the violated methods. The first defect is in the method removeDoubleEntries of the MessageBuilderHelper class. We show the code sample of that method as below:

```java
private static String removeDoubleEntries(String input) {
    Pattern sP = Pattern.compile("s*(<[^s<]+>)s*");
    ArrayList entries = new ArrayList();
    Pattern sP = Pattern.compile("s*(<[^s<]+>)s*");
    Matcher matcher = sP.matcher(input);
    while (matcher.find()) {
        entries.add(matcher.group(1));
    }
    Iterator it = entries.iterator();
    String last = (String) it.next();
    for (String curr; it.hasNext(); curr = last) {
        last = (String) it.next();
    }
    return curr;
}
```

The threats to external validity primarily include the degree to which the subject programs and CSE used are representative of true practice. The current subjects range from small-scale applications such as Java Servlets to large-scale applications when a set of interested APIs is given as input. This feature could be quite useful for APIs such as security APIs. We also showed that NEGW eb can deal with the scalability issues in processing the large number of related code samples gathered from a CSE.

5. THREATS TO VALIDITY

The threats to external validity primarily include the degree to which the subject programs and CSE used are representative of true practice. The current subjects range from small-scale applications such as Java Servlets to large-scale applications such as BCEL, Hibernate, and Columba. We used only one CSE, i.e., Google code search, which is a well-known CSE. These threats could be reduced by more experiments on wider types of subjects and by using other CSEs in future work. The threats to internal validity are instrumentation effects that can bias our results. Faults in our

4.4 Summary of the Evaluation

The major advantage of NEGW eb compared to existing approaches [1, 2, 4, 10, 15] is that NEGW eb can identify real rules due to the large number of data points used for mining the condition patterns. The number of false-positive patterns mined by NEGW eb is relatively low compared to other existing approaches as shown in our first evaluation. NEGW eb confirmed the violation of four defects that are reported in the literature and are related to neglected conditions. NEGW eb detected three new defects in the Columba application, which is a popular email client. We showed that NEGW eb can identify defects in existing open source applications when a set of interested APIs is given as input. This feature could be quite useful for APIs such as security APIs. We also showed that NEGW eb can deal with the scalability issues in processing the large number of related code samples gathered from a CSE.
NEGWWeb prototype might cause such effects. There can be errors in our inspection of source code for confirming defects. To reduce these threats, we inspected the available specifications and also call sites in source code.

6. RELATED WORK

The most related work to our NEGWWeb approach is the approach developed by Chang et al. [4] that applies frequent subgraph mining on C code to mine implicit condition rules and to detect neglected conditions. Both NEGWWeb and their approach target at the same type of defects: neglected conditions. NEGWWeb significantly differs from Chang et al.’s approach in three main aspects. First, their approach is limited on a much smaller scale of code repositories (in fact, only one project code base) than NEGWWeb, which exploits a CSE to search for billions of lines of code. Second, the scalability of their approach is heavily limited by its underlying graph mining algorithms, which are known to suffer from scalability issues, whereas NEGWWeb uses simple statistics to surface out condition rules, being much more scalable. Third, their approach reports “few apparent violations of rules” in the code base being analyzed, whereas NEGWWeb detected not only known bugs detected by other existing approaches but also previously unknown bugs.

PR-Miner developed by Li and Zhou [10] uses frequent itemset mining to extract implicit programming rules from C code and detect their violations. DynaMine developed by Livshits and Zimmermann [11] uses association rule mining to extract simple rules from software revision histories for Java code and detect bugs related to rule violations. PR-Miner or DynaMine may suffer from issues of high false positives as their rule elements are not necessarily associated with program dependencies. In addition, NEGWWeb targets at a much larger scale of code bases than PR-Miner or DynaMine.

Williams and Hollingsworth [16] incorporates an API call return value checker for C code, which checks that a value returned by an API call is tested before being used. This type of return-value testing before use falls into a subset of the types of rules being mined by NEGWWeb. Different from their tool, NEGWWeb does not require or rely on version histories, which may not include the types of bug fixing (required by their tool) related to the rules being mined. Acharya et al. [2] developed a tool to mine interface details (such as an API call’s return values on success or failure and error flags) from model-checker traces for C code, and then generate interface robustness properties for bug finding. Similar to Williams and Hollingsworth [16], Acharya et al.’s tool mines only a subset of neglected conditions (e.g., return-value testing before use) mined by NEGWWeb. In addition, as shown by Acharya et al. [2], only the interface details of 22 out of 60 POSIX API functions can be successfully mined by their tool, whereas NEGWWeb exploits a CSE to alleviate the issue by collecting relevant API call usages from the web.

Engler et al. [5] proposed a general approach for finding bugs in C code by applying statistical analysis to rank deviations from programmer beliefs inferred from source code. Their approach allows users to define rule templates. NEGWWeb follows a similar methodology to find bugs. However, beyond the general rule templates proposed in their approach, NEGWWeb’s rule templates are more specific to detecting neglected conditions around API calls and NEGWWeb incorporates various heuristics to help reduce false positives.

7. CONCLUSION

We developed a framework, called NEGWWeb, that tries to address the issue of lacking relevant data points faced by the existing static defect finding approaches that mine programming rules from source code of one or a few project code bases. NEGWWeb tries to address the preceding issue by leveraging a code search engine, which can search for related code samples in billions of lines of available open source code. NEGWWeb detects violations related to neglected conditions around individual API calls by mining condition patterns from the gathered code samples. We evaluated our framework with five open source projects and confirmed the top 25 mined condition patterns. We confirmed three known Java defects in the literature and found three new defects in a large-scale application called Columba. We also detected defects in existing open source applications that reuse a given API. In future work, we plan to extend NEGWWeb framework to the C programming language.

8. REFERENCES