The Bug Catalog of the Maven Ecosystem

Dimitris Mitropoulos* Vassilios Karakoidas* Panos Louridas* Georgios Gousios†
Diomidis Spinellis*
*Department of Management Science and Technology
†Athens University of Economics and Business
Athens, Greece
†Software Engineering Research Group
Delft University of Technology
Delft, the Netherlands
{dimitro,bkarak,louridas,dds}@aueb.gr
g.gousios@tudelft.nl

ABSTRACT
Examining software ecosystems can provide the research community with data regarding artifacts, processes, and communities. We present a dataset obtained from the Maven central repository ecosystem (approximately 265 GB of data) by statically analyzing the repository to detect potential software bugs. For our analysis we used FindBugs, a tool that examines Java bytecode to detect numerous types of bugs. The dataset contains the metrics results that FindBugs reports for every project version (a jar) included in the ecosystem. For every version we also stored specific metadata such as the jar’s size, its dependencies and others. Our dataset can be used to produce interesting research results, as we show in specific examples.

Categories and Subject Descriptors
D.2.4 [Software Engineering]: Software/Program Verification—Statistical methods; D.2.5 [Software Engineering]: Testing and Debugging—Code inspections and walk-throughs; D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement—Version control

General Terms
Static Analysis, Software Ecosystems.

Keywords
Maven Repository, FindBugs, Software Bugs.

1. INTRODUCTION
A software ecosystem can be seen as a collection of software projects, which are developed and co-evolve in the same environment [4]. Components can be interdependent and have multiple versions. Examples of such ecosystems include Python’s pypy1 (Python Package Index), Perl’s CPAN2

1http://pypy.org/
2http://www.cpan.org/

(Comprehensive Perl Archive Network), Ruby’s RubyGems3 and the Maven Central Repository.4

Maven is a build automation tool used primarily for Java projects and it is hosted by the Apache Software Foundation.5 It uses XML to describe the software project being built, its dependencies on other external modules, the build order, and required plug-ins. To build a software component, it dynamically downloads Java libraries and Maven plug-ins from the Maven central repository,6 and stores them in a local cache. The repository can be updated with new projects and also with new versions of existing projects that can depend on other versions.

To statically analyze the Maven repository we used FindBugs,7 a static analysis tool that examines bytecode to detect software bugs and has already been used in research [1, 7]. Specifically, we ran FindBugs on all the project versions of all the projects that exist in the repository to identify all bugs contained in it.

In this paper we present: a) the construction process to obtain the collection of the metrics results that the FindBugs tool produces for every project version of the repository (115,214 jars), b) our dataset and c) how researchers can use the dataset and produce meaningful results.

2. DATASET CONSTRUCTION PROCESS
The dataset construction consisted of two basic steps, namely: data cleaning and FindBugs results collection. Table 2 presents some metrics concerning the tools created for

1http://rubygems.org/
2http://mvnrepository.com/
3http://maven.apache.org/
4http://mvnrepository.com/
5http://findbugs.sourceforge.net/
Table 2: Size metrics concerning the tools created for the dataset construction process.

<table>
<thead>
<tr>
<th>Python Scripts</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lines of Code</td>
<td>1256</td>
</tr>
<tr>
<td>Code Size (KB)</td>
<td>92</td>
</tr>
</tbody>
</table>

Table 2: Size metrics concerning the tools created for the dataset construction process.

Figure 1: Distribution of version count among project population.

Data Cleaning Initially, we obtained a snapshot (January 2012) of the Maven repository and handled it locally to retrieve a list of all the names of the project versions that existed in it. In addition, we filtered out projects written in programming languages other than Java (such as Scala, Groovy, Clojure, etc.) because FindBugs analyzes only Java bytecode. Since our construction process took place at the end of 2012, several versions were released for some of the projects. Thus, we cross-checked the local versions with those online and supplemented the missing jars that have been added to the repository during this period. The statistic measurements concerning the repository can be seen in Table 1 and the distribution of version count among the selected projects is presented in Figure 1.

FindBugs Results Collection The dataset size prohibited the analysis on a single host. For this reason, we opted for distributed processing. A schematic representation of the collection of the FindBugs metrics results can be seen in Figure 2.

In particular, we created a series of processing tasks based on the JAR list from the previous step, and added them to a task queue mechanism (a RabbitMQ message broker). Then we executed twenty five workers (custom Python scripts) that checked out tasks from the queue, processed each project version and stored the results to the data repository (a MongoDB database system). The collection process lasted for approximately four and a half days.

A typical processing cycle of a worker included the following steps: after the worker spawned, it requested a task from the queue. This task contained the JAR name, which was typically a project version that was downloaded locally. First, specific JAR metadata were calculated and stored (see Section 3). Then, FindBugs was invoked by the worker and its results were also stored in the data repository. Note that before invoking FindBugs, the worker checked if the JAR is valid in terms of implementation. For instance, for every JAR the worker checked if there were any .class files before invoking FindBugs. When the task was completed the queue was notified and the next task was requested.

When the data collection was completed, we ran some tests to check the validity of the results. A common issue that we discovered was the out-of-memory crashes of Findbugs, which demanded the repetition of the process for the corresponding JARs, with the appropriate settings in the Java Runtime Environment (JRE).

3. DATASET CONTENTS

As we mentioned earlier our data was stored in a MongoDB database. FindBugs separates software bugs into nine categories (see Table 3) and reports bug collections that include all the bugs discovered in a JAR file. For every registered bug,
there are numerous accompanying features like the class, the method and the line that the bug was found. FindBug’s results, also include additional information like the number of classes included in the examined JAR and others. FindBugs though, outputs its results in XML format. Hence, all the data were converted to the JSON format by mapping all XML elements to JSON objects.

As we mentioned in Section 2, our workers calculated and stored specific metadata together with the FindBugs results. Such metadata included the JAR’s size (in terms of bytecode), its dependencies (derived from the pom.xml file), and a number that represented the ordinal version number of the release (see also Table 4). This number was derived from an XML file that accompanies every project in the Maven repository called maven-metadata.xml. The following listing shows the format of the information we collected (note that the results of FindBugs are too large to show. To see a complete instance please visit our GitHub repository — see Section 7):

```json
{ "JarMetadata": {
    "version": "1.0.0",
    "version_order": "1",
    "jar_size": "34768",
    "dependencies": [
        { "version": "2.0",
        "groupId": "org.apache.maven",
        "artifactId": "maven-project"
        }, /* other dependencies */
    ],
    "group_id": "org.apache.myfaces.buildtools",
    "jar_filename": "myfaces-jdev-plugin-1.0.0.jar",
    "artifact_id": "myfaces-jdev-plugin",
},
"BugCollection": { /* FindBugs data */ }
}
```

4. HARNESSING OUR DATASET

Since MongoDB provides a rich query interface, it was easy to create simple scripts to find out how software bugs are distributed among the repository (see Figure 3).

Furthermore, we have created a series of scripts to exhibit how the dataset can be used to capture correlations regarding the evolution of software bugs. First, based on the dataset we produced some metadata that contained the number of bugs per category in each project version. Based on these metadata we estimated the relation between bugs and time (see Table 5). Specifically, we calculated the Spearman correlations between the defects count and the ordinal version number across all projects. The zero tendency that can be seen on Table 5 applies to all versions of all projects together.

In addition, we explored the relation between defects with the size of a project version, measured by the size of its JAR file by carrying out correlation tests between the size and the defect counts for each project and version. The results can be seen in Table 6.

Figure 4 presents the pairwise correlations between all bug categories. To establish these correlations, we calculated the correlations between the number of distinct bugs that appeared in a project throughout its lifetime.

Research concerning the examination of specific bugs can also be facilitated by our dataset. For instance, we have examined the characteristics of security bugs in a previous paper [5] based on this dataset.

5. LIMITATIONS

Limitations concerning our dataset involve some JARS that were in the initial JAR list created during data cleaning but when the FindBugs result collection was performed, they

<table>
<thead>
<tr>
<th>Category</th>
<th>Spearman Correlation</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Security</td>
<td>0.04</td>
<td>&lt; 0.05</td>
</tr>
<tr>
<td>Malicious Code</td>
<td>0.03</td>
<td>&lt; 0.05</td>
</tr>
<tr>
<td>Style</td>
<td>0.03</td>
<td>&lt; 0.05</td>
</tr>
<tr>
<td>correctness</td>
<td>0.04</td>
<td>&lt; 0.05</td>
</tr>
<tr>
<td>Bad Practice</td>
<td>0.03</td>
<td>&lt; 0.05</td>
</tr>
<tr>
<td>i18n</td>
<td>0.06</td>
<td>&lt; 0.05</td>
</tr>
<tr>
<td>Performance</td>
<td>(0.01)</td>
<td>0.07</td>
</tr>
<tr>
<td>Experimental</td>
<td>0.09</td>
<td>&lt; 0.05</td>
</tr>
</tbody>
</table>

Table 5: Correlations between version and defects count.
were not online.

A threat to the internal validity of our dataset construction process could be the false alarms of the FindBugs tool [1]. Specifically, reported security bugs may not be applicable to an application’s typical use context. For instance, FindBugs could report an SQL injection vulnerability in an application that receives no external input. In this particular context, this would be a false positive alarm.

A disadvantage of our dataset is that as projects evolve the dataset gets older. Also, even if we checked for new project versions that could have been added to the repository after the data cleaning and before the results collection, we did not checked for new projects added during this time frame.

6. RELATED WORK

The Maven ecosystem has been previously analyzed by Raemaekers et al. [6] to produce the Maven dependency dataset. Apart from basic information like individual methods, classes, packages and lines of code for every JAR, this dataset also includes a database with all the connections between the aforementioned elements. Our work differs from this research because it reports all bugs coming from the output of a static analysis tool, for each JAR contained in the Maven repository.

7. CONCLUSIONS

In this paper, we have presented a dataset that contains for every JAR of the Maven central repository, all the software bugs that it contains among with some other metadata mentioned in Section 4. We have also shown how our data can be used to extract results concerning software evolution.

Note that our data collection process can be duplicated in order to collect metrics results from other tools that examine Java programs like CJKM⁹ that calculates Chidamber and Kemerer object-oriented metrics [2], Java Pathfinder¹⁰ that performs model checking [3] on Java bytecode programs, and others. To achieve this, instead of invoking FindBugs, our worker can invoke such tools.

The complete set of our data and source code can be found at https://github.com/bkarak/data_paper_msr2014.

8. REFERENCES


Table 6: Correlations between JAR size and defects count.

<table>
<thead>
<tr>
<th>Category</th>
<th>Spearman Correlation</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Security</td>
<td>0.19</td>
<td>≪ 0.05</td>
</tr>
<tr>
<td>Malicious Code</td>
<td>0.65</td>
<td>≪ 0.05</td>
</tr>
<tr>
<td>Style</td>
<td>0.68</td>
<td>≪ 0.05</td>
</tr>
<tr>
<td>Correctness</td>
<td>0.51</td>
<td>≪ 0.05</td>
</tr>
<tr>
<td>Bad Practice</td>
<td>0.67</td>
<td>≪ 0.05</td>
</tr>
<tr>
<td>MT Correctness</td>
<td>0.51</td>
<td>≪ 0.05</td>
</tr>
<tr>
<td>i18n</td>
<td>0.53</td>
<td>≪ 0.05</td>
</tr>
<tr>
<td>Performance</td>
<td>0.63</td>
<td>≪ 0.05</td>
</tr>
<tr>
<td>Experimental</td>
<td>0.36</td>
<td>≪ 0.05</td>
</tr>
</tbody>
</table>

Figure 4: Correlation matrix plot for bug categories. Blue color indicates positive correlation. The darker the color (and the more acute the ellipsis slant), the stronger the correlation.

Table 6: Correlations between JAR size and defects count.

<table>
<thead>
<tr>
<th>Category</th>
<th>Spearman Correlation</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Security</td>
<td>0.19</td>
<td>≪ 0.05</td>
</tr>
<tr>
<td>Malicious Code</td>
<td>0.65</td>
<td>≪ 0.05</td>
</tr>
<tr>
<td>Style</td>
<td>0.68</td>
<td>≪ 0.05</td>
</tr>
<tr>
<td>Correctness</td>
<td>0.51</td>
<td>≪ 0.05</td>
</tr>
<tr>
<td>Bad Practice</td>
<td>0.67</td>
<td>≪ 0.05</td>
</tr>
<tr>
<td>MT Correctness</td>
<td>0.51</td>
<td>≪ 0.05</td>
</tr>
<tr>
<td>i18n</td>
<td>0.53</td>
<td>≪ 0.05</td>
</tr>
<tr>
<td>Performance</td>
<td>0.63</td>
<td>≪ 0.05</td>
</tr>
<tr>
<td>Experimental</td>
<td>0.36</td>
<td>≪ 0.05</td>
</tr>
</tbody>
</table>