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
{dimitro,bkarak,louridas,dds}@aueb.gr
†Software Engineering Research Group
†Delft University of Technology
Delft, the Netherlands
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 265GB 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 PyPY[1] (Python Package Index), Perl's CPAN[2] (Comprehensive Perl Archive Network), Ruby's RubyGems[3] 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.

Table 1: Descriptive statistics measurements for the Maven repository.

Measurement	Value
Projects	17,505
Versions (total)	115,214
Min (versions per project)	1
Max (versions per project)	338
Mean (versions per project)	6.58
Median (versions per project)	3
Range (over versions)	337
1^{st} Quartile (over versions)	1
3^{rd} Quartile (over versions)	8

2. DATASET CONSTRUCTION PROCESS

The dataset construction consisted of two basic steps, namely: *data cleaning* and *FindBugs results collection*. Ta-

[1] http://pypy.org/
[2] http://www.cpan.org/
[3] http://rubygems.org/
[4] http://mvnrepository.com/
[5] http://maven.apache.org/
[6] http://mvnrepository.com/
[7] http://findbugs.sourceforge.net/

MSR'14, May 31 – June 1, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2863-0/14/05...$15.00
http://dx.doi.org/10.1145/2597073.2597123

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

Python Scripts	14
Lines of Code	1256
Code Size (KB)	92

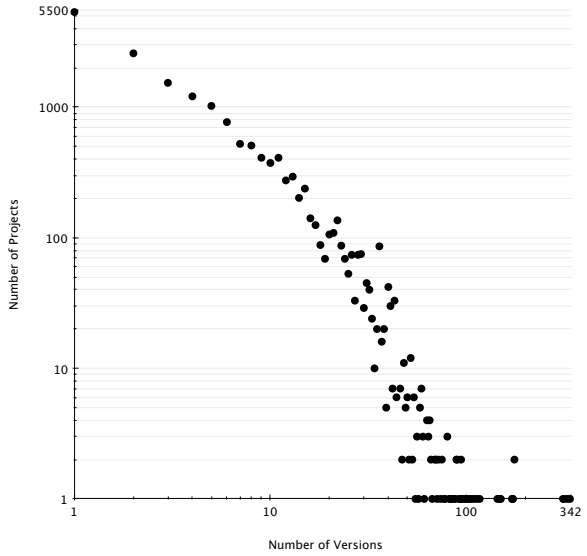

Figure 1: Distribution of version count among project population.

ble 2 presents some metrics concerning the tools created for the process.

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[8] 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.

[8]http://mirrors.ibiblio.org/maven2/

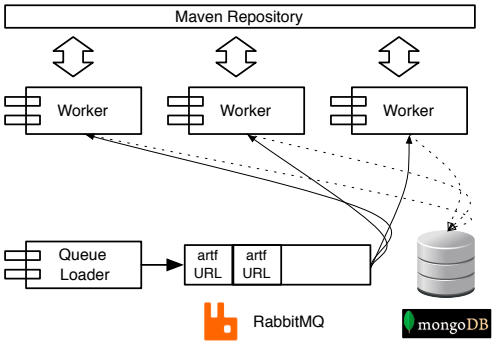

Figure 2: The data processing architecture.

Table 3: Bug categorization according to FindBugs.

Category	Description
Bad Practice	Violations of recommended and essential coding practice.
Correctness	Involves coding misting a way that is particularly different from the other bug sakes resulting in code that was probably not what the developer intended.
Experimental	Includes unsatisfied obligations. For instance, forgetting to close a file.
Internationalization (i18n)	Indicates the use of non-localized methods.
Multi-Threaded (MT) Correctness	Thread synchronization issues.
Performance	Involves inefficient memory usage allocation, usage of non-static classes.
Style	Code that is confusing, or written in a way that leads to errors.
Malicious Code	Involves variables or fields exposed to classes that should not be using them.
Security	Involves input validation issues, unauthorized database connections and others.

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):

```
{"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.

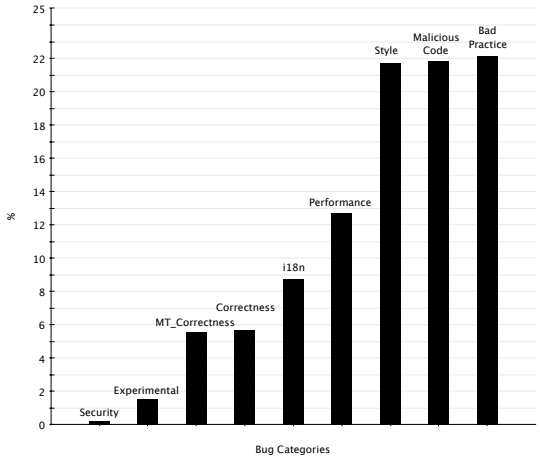

Figure 3: Bug percentage in Maven repository.

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.

Table 7 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 out dataset involve some JARs that were in the initial JAR list created during data cleaning but when the FindBugs result collection was performed, they 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

Table 4: JAR metadata description.

`version`	The version of the project in the repository.
`version_order`	The ordinal version number of the release.
`jar_size`	The size of the JAR file.
`dependencies`	List of all dependencies for the project.
`group_id`	The group ID of the project in Maven repository.
`jar_filename`	JAR's filename.
`artifact_id`	The artifact ID of the project in Maven repository.

Table 5: Correlations between version and defects count.

Category	Spearman Correlation	p-value
Security	0.04	< 0.05
Malicious Code	0.03	$\ll 0.05$
Style	0.03	$\ll 0.05$
Correctness	0.04	$\ll 0.05$
Bad Practice	0.03	$\ll 0.05$
MT Correctness	0.09	$\ll 0.05$
i18n	0.06	$\ll 0.05$
Performance	*(0.01)*	0.07
Experimental	0.09	$\ll 0.05$

Table 7: Correlations between bug categories.

Category	Security	Malicious Code	Style	Bad Practice	Correctness	MT Correctness	Performance	i18n	Experimental
Security	1.00	0.20	0.22	0.27	0.20	0.33	0.20	0.31	0.32
Malicious Code	0.20	1.00	0.63	0.63	0.56	0.56	0.62	0.60	0.48
Style	0.22	0.63	1.00	0.68	0.62	0.63	0.66	0.58	0.46
Bad Practice	0.27	0.63	0.68	1.00	0.54	0.58	0.64	0.63	0.51
Correctness	0.20	0.56	0.62	0.54	1.00	0.56	0.55	0.52	0.52
MT Correctness	0.33	0.56	0.63	0.58	0.56	1.00	0.56	0.56	0.49
Performance	0.20	0.62	0.66	0.64	0.55	0.56	1.00	0.60	0.52
i18n	0.31	0.60	0.58	0.63	0.52	0.56	0.60	1.00	0.60
Experimental	0.32	0.48	0.46	0.51	0.52	0.49	0.52	0.60	1.00

Table 6: Correlations between JAR size and defects count.

Category	Spearman Correlation	p-value
Security	0.19	$\ll 0.05$
Malicious Code	0.65	$\ll 0.05$
Style	0.68	$\ll 0.05$
Correctness	0.51	$\ll 0.05$
Bad Practice	0.67	$\ll 0.05$
MT Correctness	0.51	$\ll 0.05$
i18n	0.53	$\ll 0.05$
Performance	0.63	$\ll 0.05$
Experimental	0.36	$\ll 0.05$

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 CJKM[9] that calculates Chidamber and Kemerer object-oriented metrics [2], *Java Pathfinder*[10] 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`.

[9] `http://www.spinellis.gr/sw/ckjm/`

[10] `http://babelfish.arc.nasa.gov/trac/jpf/wiki`

8. ACKNOWLEDGMENTS

This research has been co-financed by the European Union (European Social Fund—ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF)—Research Funding Program: Thalis—Athens University of Economics and Business—Software Engineering Research Platform.

9. REFERENCES

[1] N. Ayewah and W. Pugh. The google FindBugs fixit. In *Proceedings of the 19th international symposium on Software testing and analysis*, ISSTA '10, pages 241–252, New York, NY, USA, 2010. ACM.

[2] S. R. Chidamber and C. F. Kemerer. Towards a metrics suite for object oriented design. In *Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '91, pages 197–211, New York, NY, USA, 1991. ACM.

[3] E. M. Clarke, E. A. Emerson, and J. Sifakis. Model checking: Algorithmic verification and debugging. *Commun. ACM*, 52(11):74–84, Nov. 2009.

[4] M. Lungu and M. Lanza. The small project observatory: A tool for reverse engineering software ecosystems. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 289–292, New York, NY, USA, 2010. ACM.

[5] D. Mitropoulos, V. Karakoidas, P. Louridas, G. Gousios, and D. Spinellis. Dismal code: Studying the evolution of security bugs. In *Proceedings of the LASER Workshop 2013, Learning from Authoritative Security Experiment Results*, pages 37–48. Usenix Association, Oct. 2013.

[6] S. Raemaekers, A. v. Deursen, and J. Visser. The maven repository dataset of metrics, changes, and dependencies. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 221–224, Piscataway, NJ, USA, 2013. IEEE Press.

[7] J. Spacco, D. Hovemeyer, and W. Pugh. Tracking defect warnings across versions. In *Proceedings of the 2006 international workshop on Mining software repositories*, MSR '06, pages 133–136, New York, NY, USA, 2006. ACM.