# Dismal Code: Studying the Evolution of Security Bugs

Dimitris Mitropoulos, Vassilios Karakoidas, Panos Louridas
*Department of Management Science and Technology,*
*Athens University of Economics and Business*
`dimitro@aueb.gr, bkarak@aueb.gr, louridas@aueb.gr`

Georgios Gousios
*Software Engineering Research Group, Delft University of Technology*
`G.Gousios@tudelft.nl`

Diomidis Spinellis
*Department of Management Science and Technology,*
*Athens University of Economics and Business*
`dds@aueb.gr`

## Abstract

**Background.** Security bugs are critical programming errors that can lead to serious vulnerabilities in software. Such bugs may allow an attacker to take over an application, steal data or prevent the application from working at all.
**Aim.** We used the projects stored in the Maven repository to study the characteristics of security bugs individually and in relation to other software bugs. Specifically, we studied the evolution of security bugs through time. In addition, we examined their persistence and their relationship with a) the size of the corresponding version, and b) other bug categories.
**Method.** We analyzed every project version of the Maven repository by using FindBugs, a popular static analysis tool. To see how security bugs evolve over time we took advantage of the repository's project history and dependency data.
**Results.** Our results indicate that there is no simple rule governing the number of security bugs as a project evolves. In particular, we cannot say that across projects security-related defect counts increase or decrease significantly over time. Furthermore, security bugs are not eliminated in a way that is particularly different from the other bugs. In addition, the relation of security bugs with a project's size appears to be different from the relation of the bugs coming from other categories. Finally, even if bugs seem to have similar behaviour, severe security bugs seem to be unassociated with other bug categories.
**Conclusions.** Our findings indicate that further research should be done to analyze the evolution of security bugs. Given the fact that our experiment included only Java projects, similar research could be done for another ecosystem. Finally, the fact that projects have their own idiosyncrasies concerning security bugs, could help us find the common characteristics of the projects where security bugs increase over time.

## 1 Introduction

A security bug is a programming error that introduces a potentially exploitable weakness into a computer system [34]. This weakness could lead to a security breach with unfortunate consequences in different layers, like databases, native code, applications, libraries and others. Despite the significant effort to detect and eliminate such bugs, little attention has been paid to study them in relation to software evolution [26]. One of the most common approaches to identify security bugs is *static analysis* [6]. This kind of analysis involves the inspection of the program's source or object code without executing it.

In this paper we present how we used a large software ecosystem to analyse the relationship of different types of security vulnerabilities to evolving software packages. For our research we used *FindBugs*,[1] a static analysis tool that examines bytecode to detect software bugs and has already been used in research [1, 19, 36]. Specifically, we ran FindBugs on all the project versions of all the projects that exist in the *Maven Central Repository*[2] (approximately 265GB of data—see Section 3.2). Then we observed the changes that involved the security bugs and their characteristics.

We chose to focus our study on security bugs rather than other types of software defects. This is because compared to other bug categories, failures due to security bugs have two distinct features: they can severely affect an organization's infrastructure [33], and they can cause significant financial damage to an organization [39, 2]. Specifically, whereas a software bug can cause a software artifact to fail, a security bug can allow a malicious user to alter the execution of the entire application for his or her own gain. In this case, such bugs could give rise to a wide range of security and privacy issues, like the access of sensitive information, the destruction or modification of data, and denial of service. Moreover, security bug disclosures lead to a negative and significant change in market value for a software vendor [38]. Hence, one of the basic pursuits in every new software release should be to mitigate such bugs.

The motivation behind our work was to validate whether programmers care for the risk posed by security bugs when they release a new version of their software. In addition, we wanted to investigate other critical features associated with such vulnerabilities like the persistence of a bug; in essence, to see whether critical bugs stay unresolved for a long time. Also, we wanted to elaborate more on the relation of security bugs with other bug categories. In the same manner, we tried to examine the relationship between the size of a project release and the number of security bugs that it contains, knowing the that research has produced contradictory results on this issue [35, 28, 13]. Finally, we examined the Maven ecosystem as a whole from a security perspective. Its structure gave us the opportunity to see if a project version that is a dependency of a large number of others contains a low rate of security bugs.

In this work we:

- Analyze how security bugs found through static analysis evolve over time. To achieve this, we inspected all releases of every project. Our hypothesis is that security bugs should decrease as a project evolves, for they form critical issues, which developers should eliminate.

- Examine security bug persistence across releases. We expect that security bugs should be eliminated earlier than other bugs.

- Study the relation between security bugs and a project release's size. Our hypothesis is that security bugs are proportional to a project release's size (defined in terms of bytecode size).

- Examine the correlation of security bugs with other bug categories. Our hypothesis is that security bugs appear together with bugs that are related with performance, coding practices, and product stability.

In the rest of this paper we outline related work (Section 2), describe the processing of our data and our experiment (Section 3), present and discuss the results we obtained (Section 4), and end up with a conclusion and directions for future work (Section 5).

## 2 Related Work

There are numerous methods for mining software repositories in the context of software evolution [20]. In this section we focus on the ones that highlight the relationship between software bugs and evolution and try to extract useful conclusions.

*Refactoring identification* through software evolution is an approach used to relate refactorings with software bugs. Weißgerber et al. found that a high ratio of refactorings is usually followed by an increasing ratio of bug reports [40]. In addition, they indicated that software bugs are sometimes introduced after an incomplete refactoring [12]. Ratzinger et al. [31] showed that the number of bugs decreases, when the number of refactorings increases. Finally, Kim M. et al. [22] indicated that API-level refactorings aid bug fixes.

*Micro patterns*, proposed by Kim et al. [24] detect bug-prone patterns among source code. Micro patterns describe programming idioms like inheritance, data management, immutability and others. The approach involved the examination of all revisions of three open-source projects to extract bug introduction rates for each pattern. Gil et al. [11] analysed the prevalence of micro patterns across five Sun JDK versions to conclude that pattern prevalence tends to be the same in software collections.

*Querying techniques* are used to answer a broad range of questions regarding the evolution history of a project [17]. Bhattacharya et al. [4, 3] proposed a framework that is based on recursively enumerable languages. The framework can correlate software bugs with developers in various ways. For instance, return the list of bugs fixed by a specific developer. Fischer et al. [10] proposed an approach for populating a release history database that combines code information with bug tracking data. In this way, a developer can couple files that contain common bugs, estimate code maturity with respect to the bugs, etc. The "Ultimate Debian Database" [29] is an SQL-based framework that integrates information about the Debian project from various sources to answer queries related to software bugs and source code.

D'Ambros et al. have used *bug history analysis* to detect the critical components of a project [7]. This is done by using an evolutionary meta-model [8]. The same approach was also used by Zimmermann et al. [42] to check the correlation of bugs with software properties like code complexity, process quality and others and to predict future properties.

The evolution of software artifacts has also been analysed to *reduce the false alarms* of the various static analysis tools. To achieve this, Spacco et al. [36] introduced *pairing* and *warning signatures*. In the former, they tried to pair sets of bugs between versions in order to find similar patterns. In the latter, they computed a signature for every bug. This signature contained elements like the name of the class where the bug was found, the method and others. Then they searched for similar signatures between versions. In their research they studied the evolution of 116 sequential builds of the Sun Java Sevelopment Kit (JDK). Their findings indicated that high priority bugs are fixed over time. To improve the precision of bug detection, Kim et al. [23] proposed a history-based warning prioritization algorithm by mining the history of bug-fixes of three different projects. Working towards the same direction, Heckman et al. [15, 14] have introduced benchmarks that use specific correlation algorithms and classification techniques to evaluate alert prioritization approaches.

Lu et al. [25] studied the *evolution of file-system code*. Specifically, they analysed the changes of Linux file-system patches to extract bug patterns and categorize bugs based on their impact. Their findings indicated that the number of file-system bugs does not die down over time. By categorizing bugs they also showed the frequency of specific bugs in specific components.

Completing the above approaches, our work focuses on the subset of security bugs. Focusing on such bugs is not a new idea. Ozment and Schechter [30] examined the code base of the OpenBSD operating system to determine whether its security is increasing over time. In particular, they measured the rate at which new code has been introduced and the rate at which defects have been reported over a 7.5 year period and fifteen releases. Even though the authors present statistical evidence of a decrease in the rate at which vulnerabilities are being reported, defects seem to appear persistent for a period of at least 2.6 years. Massacci et al. [27] observed the evolution of software defects by examining six major versions of Firefox. To achieve this they created a database schema that contained information coming from the "Mozilla Firefox-related Security Advisories" (MFSA) list,[3] Bugzilla entries and others. Their findings indicated that security bugs are persistent over time. They also showed that there are many web users that use old versions of Firefox, meaning that old attacks will continue to work. Zaman et al. [41] focused again on Firefox to study the relation of security bugs with performance bugs. This was also done by analysing the project's Bugzilla. Their research presented evidence that security bugs require more experienced developers to be fixed. In addition, they suggested that security bug fixes are more complex than the fixes of performance and other bugs. Shahzad et al. [34] analysed large sets of vulnerability data-sets to observe various features of the vulnerabilities that they considered critical. Such features were the functionality and the criticality of the defects. Their analysis included the observation of vulnerability disclosures, the behavior of hackers in releasing exploits for vulnerabilities, patching and others. In their findings they highlighted the most exploited defects and showed that the percentage of remotely exploitable vulnerabilities has gradually increased over the years. Finally, Edwards et al. [9] have recently conducted a study similar to ours in which they have considered only four projects. Their results demonstrate that the number of exploitable bugs does not always improve with each new release and that the rate of discovery of exploitable bugs begins to drop three to five years after the initial release.

## 3 Methodology

Our experiment involved the collection of the metric results of the FindBugs tool. Before and during the experiment, we performed a number of filters on the data coming from the Maven repository, for reasons that we will describe below.

### 3.1 Experiment

The goal of our experiment was to retrieve all the bugs that FindBugs reports, from all the project versions existing on the Maven repository (in the Maven repository, versions are *actual* releases). The experiment involved four entities: a number of *workers* (a custom Python script), a *task queue* mechanism (RabbitMQ—version 3.0.1), a *data repository* (MongoDB—version 2.2), and the *code repository*, which in our case it was the public Maven repository.

Maven is a build automation tool used primarily for Java projects and it is hosted by the Apache Software Foundation. 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, and stores them in a local cache. The repository can be updated with new projects and also with new versions of existing projects.

First, we scanned the Maven repository for appropriate JARs and created a list that included them. We discuss the JAR selection process in the next section. With the JAR

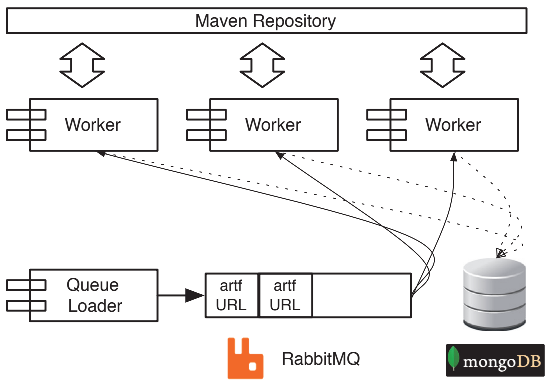

Figure 1: The data processing architecture.

list at hand, we created a series of processing tasks and added them to the task queue. Then we executed twenty five (Unix-based) workers that checked out tasks from the queue, processed the data and stored the results to the data repository.

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. Such metadata included its size, its dependencies, and a number that represented the chronological order of the release. This order was derived from an XML file that accompanies every project in the Maven repository called *maven-metadata.xml*. Then, FindBugs was invoked by the worker and its results were also stored in the data repository. When the task was completed the queue was notified and the next task was requested. This process was executed for all the available JARs in the task queue. A schematic representation of the data processing architecture can be seen in Figure 1.

## 3.2 Data Provenance

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. A project version can be uniquely identified by the triplet: *group id*, *artifact id* and *version*.

FindBugs works by examining the compiled Java virtual machine bytecodes of the programs it checks, using the bytecode engineering library (BCEL). To detect a bug, FindBugs uses various formal methods like *control flow* and *data flow analysis*. It has also other detectors that employ *visitor patterns* over classes and methods by using *state machines* to reason about values stored in variables or on the stack. Since FindBugs analyses applications written in the Java programming language, and the Maven repository hosts projects from languages other than Java such as Scala, Groovy, Clojure, etc., we filtered out such projects by performing a series of checks in the repository data and metadata.

In addition, we implemented a series of audits in the worker scripts that checked if the JARs are valid in terms of implementation. For instance, for every JAR the worker checked if there were any *.class* files before invoking FindBugs. After the project filtering, we narrowed down our data set to 17,505 projects with 115,214 versions. Table 1 summarises the data set information and provides the basic descriptive statistic measurements. The distribution of version count among the selected projects is presented in Figure 2.

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 |

The statistical measurements presented in Table 1 indicate that we have 17,505 projects and the data set's median is 3, which means that almost 50% (8,753 projects) of the project population have 1 to 3 versions. In general, most projects have a few number of versions, there are some projects with ten versions and only a few with hundreds of versions. The maximum number of versions for a project is 338. The $3^{rd}$ quartile measurement also indicated that 75% (13,129) of the projects have a maximum of 8 versions.

## 3.3 Threats to Validity

A threat to the internal validity of our experiment could be the false alarms of the FindBugs tool [1, 18]. False positives and negatives of static analysis tools and how they can be reduced is an issue that has already been discussed in the literature (see Section 2). In addition, reported security bugs may not be applicable to an application's typical use context. For instance, FindBugs could report an SQL injection vulnerability [32] in an application that receives no external input. In this particular context, this would be a false positive alarm.

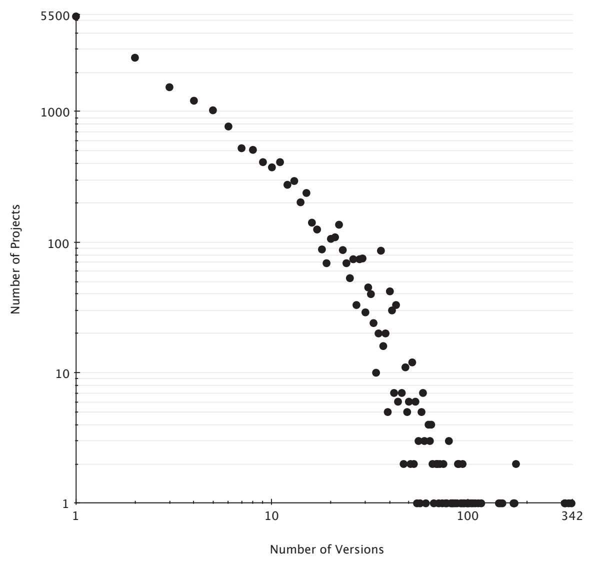

Figure 2: Distribution of version count among project population.

Furthermore, given that our analysis is done on open-source projects written in the Java programming language and hosted on Maven, a threat to the external validity of our work is the fact that our results may not be applicable to other programming languages, ecosystems, and development cultures. In particular, a large class of security problems such as buffer overflows [21] do not apply in our study since Java enforces bound checking at runtime.

## 4 Results and Analysis

Our findings can be analysed at two levels. First, we discuss some primary observations concerning the security bugs of the Maven repository as a whole. Then, we provide a comprehensive analysis of the results and highlight our key findings.

### 4.1 Overview and Initial Results

FindBugs separates software bugs into nine categories (see Table 2). Two of them involve security issues: *Security* and *Malicious Code*. From the total number of releases, 4,353 of them contained at least one bug coming from the first category and 45,559 coming from the second.

Our first results involve the most popular bugs in the Maven repository. Figure 3 shows how software bugs are distributed among the repository. Together with the *Bad Practice* bugs and the *Style* bugs, security bugs (the sum of the *Security* and *Malicious Code* categories - 0.21% + 21.81%) are the most popular in the repository ($\geq 21\%$ each). This could be a strong indication that programmers write code that implements the required functionality without considering its many security aspects; an issue that has already been reported in literature [37].

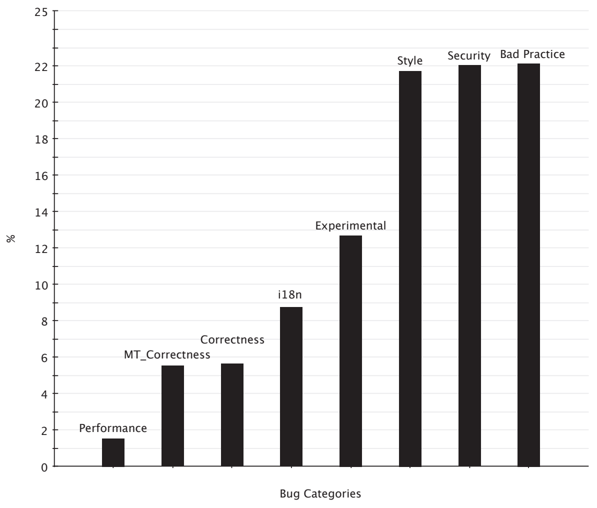

Figure 3: Bug percentage in Maven repository.

Another observation involves bugs that we could call *Security High* and they are a subset of the *Security* category. Such bugs are related to vulnerabilities that appear due to the lack of user-input validation and can lead to damaging attacks like SQL injection and Cross-Site Scripting [32]. To exploit such vulnerabilities, a malicious user does not have to know anything about the application internals. For almost all the other security bugs (coming from the *Malicious Code* category and the rest of the *Security* category bugs), another program should be written to incorporate references to mutable objects, access non-final fields, etc. Also, as bug descriptions indicate,[4] if an application has bugs coming from the *Security High* category, it might have more vulnerabilities that FindBugs doesn't report. Table 3 presents the number of releases where at least one of these bugs exists. In essence, 5,501 releases ($\approx 4,77\%$), contained at least one severe security bug. Given the fact that other projects include these versions as their dependencies, they are automatically rendered vulnerable if they use the code fragments that include the defects. The remaining bugs of the *Security* category are grouped together with the bugs of the *Malicious Code* category in another subcategory that we call *Security Low*. This category contains for the most part, bugs that imply violations of good OOP (object-oriented programming) design (i.e. keeping variables private to classes and others). The above categorization was done specifically to point out the behaviour of bugs that currently top the corresponding lists of most

Table 2: Bug categorisation 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. |

Table 3: Number of project releases that contain at least one "Security High" bug.

| Bug Description | Number of Project Releases |
|---|---|
| HRS: HTTP cookie formed from untrusted input | 151 |
| HRS: HTTP response splitting vulnerability | 1,579 |
| PT: absolute path traversal in servlet | 103 |
| PT: relative path traversal in servlet | 57 |
| SQL: non-constant string passed to execute method on an SQL statement | 1,875 |
| SQL: a prepared statement is generated from a non-constant String | 1,486 |
| XSS: JSP reflected cross site scripting vulnerability | 18 |
| XSS: Servlet reflected cross site scripting vulnerability in error page | 90 |
| XSS: Servlet reflected cross site scripting vulnerability | 142 |

security providers.[5]

*Linus's Law* states that "given enough eyeballs, all bugs are shallow". In a context like this, we expect that the project versions that are dependencies to many other projects would have a small number of security bugs. To examine this variation of the Linus's Law and highlight the *domino effect* [39] we did the following: during the experiment we retrieved the dependencies of every version. Based on this information we created a graph that represented the snapshot of the Maven repository. The nodes of the graph represented the versions and the vertices their dependencies. The graph was not entirely accurate. For instance, if a dependency was pointing only to a project (and not to a specific version), we chose to select the latest version found on the repository. Also, this graph is not complete. This is because there were missing versions. From the 565,680 vertices, 191,433 did not point to a specific version while 164,234 were pointing to missing ones. The graph contained 80,354 nodes. Obviously, the number does not correspond to the number of the total versions (see Section 3.2). This is because some versions did not contain any information about their dependencies so they are not represented in the graph. After creating the graph, we ran the PageRank algorithm [5] on it and retrieved all PageRanks for every node. Then we examined the security bugs of the fifty most popular nodes based on their PageRank. Contrary to Linus's Law, thirty three of them contained bugs coming from the *Security Low* subcategory, while two of them contained *Security High* bugs. Twenty five of them were latest versions at the time. This also highlights the domino effect.

## 4.2 Analysis

Here, we present our key findings concerning the evolution of security bugs.

### 4.2.1 How Security Bugs Evolve Over Time

The relation between bugs and time can be traced from the number of bugs per category in each project version. We can then calculate the Spearman correlations between the defects count and the ordinal version number across all projects to see if bigger versions relate to higher or lower defect counts. The results are shown in Table 4. Although the tendency is for defect counts to increase, this tendency is extremely slight.

The zero tendency applies to all versions of all projects together. The situation might be different in individual projects. We therefore performed Spearman correlations between bug counts and version ordinals in all projects we examined. These paint a different picture from the above table, shown in Figure 4. The spike in point zero is explained by the large number of projects for which no correlation could be established—note that the scale is logarithmic. Still, we can see that there were projects where a correlation could be established, either positive or negative. The *Security High* category is particularly bimodal, but this is explained by the small number of correlations that could be established, nine in total.

Overall, Table 4 and Figure 4 suggest that **we cannot say that across projects defect counts increase or decrease significantly across time**. In individual projects, however, defect counts can have a strong upwards or downwards tendency. There may be no such thing as a "project" in general, only particular projects with their own idiosyncrasies, quality features, and coding practices.

Another take on this theme is shown in Figure 5, which presents a histogram of the changes of different bug counts in project versions. In most cases, a bug count does not change between versions; but when it does change, it may change upwards or downwards. Note also the spectacular changes of introducing or removing thousands of defects; this may be the result of doing and undoing a pervasive code change that runs foul of some bug identification rule.

Table 4: Correlations between version and defects count.

| Category | Spearman Correlation | $p$-value |
|---|---|---|
| Security High | 0.08 | $\ll 0.05$ |
| Security Low | 0.02 | $\ll 0.05$ |
| 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 | $\ll 0.05$ |

### 4.2.2 Persistence of Security Bugs

To examine the relation between the persistence of different kinds of bugs, and of security bugs in particular, we used as a persistence indicator the number of versions a bug remains open in a project. To "tag" a bug we created a bug identifier by using the type of the bug, the method name and the class name in which the bug was found in. We chose not to use the line number of the location of the bug since it could change from version to version and after a possible code refactoring. We grouped the persistence numbers by bug categories and then performed a Mann-Whitney $U$ [16] test among all bug category pairs. The results are presented in Table 6 (at the end of this paper). Cells in brackets show pairs where no statistically significant difference was found.

In general, although the average number of versions bugs in different bug categories that remained open was statistically different in many cases, the difference is not spectacular. **In all cases a bug persists on average between two and three versions**, with the difference being in the decimal digits.

### 4.2.3 The Relation of Defects with the size of a JAR

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, all statistically significant ($p \ll 0.05$) can be seen in Table 5. **The** *Security High* **category stands out by having a remarkably lower effect than the other categories**, even *Security Low* that nearly tops the list. As we mentioned earlier, bugs that belong to the *Security High* category are related to user-input validation issues. Hence, even if a programmer adds code to a new version, if this code does not require user input, the possibility of such bug is minimal.

Table 5: Correlations between JAR size and defects count.

| Category | Spearman Correlation | $p$-value |
|---|---|---|
| Security High | 0.19 | $\ll 0.05$ |
| Security Low | 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 | $\ll 0.05$ |

### 4.2.4 Security Bugs VS Other Bug Categories

To see whether bugs flock together we performed pairwise correlations between all bug categories. We calculated the correlations between the number of distinct bugs that appeared in a project throughout its lifetime, see Figure 6. We found significant, but not always strong, correlations between all pairs. In general, the *Security*

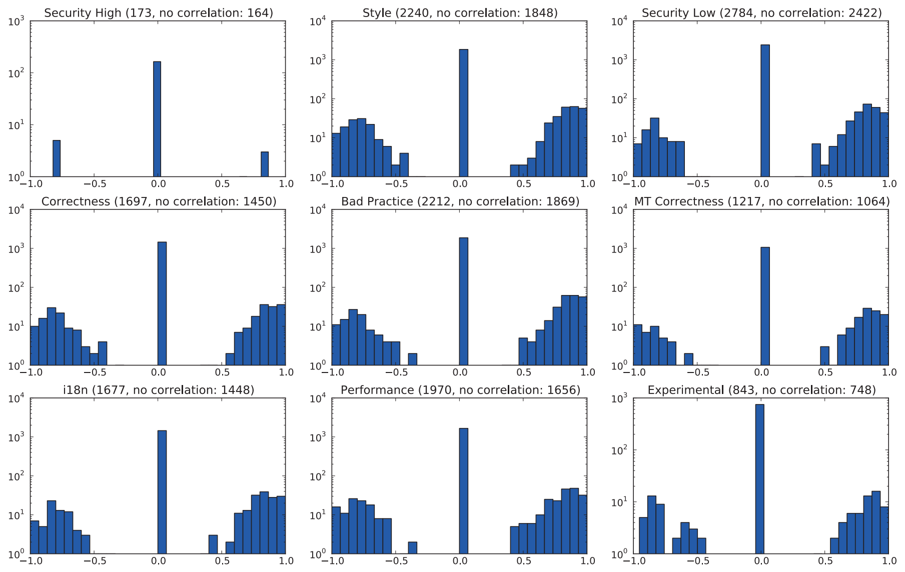

Figure 4: Histograms of correlations between bug counts and version ordinals per project. In brackets the total population size and the number of no correlation instances.

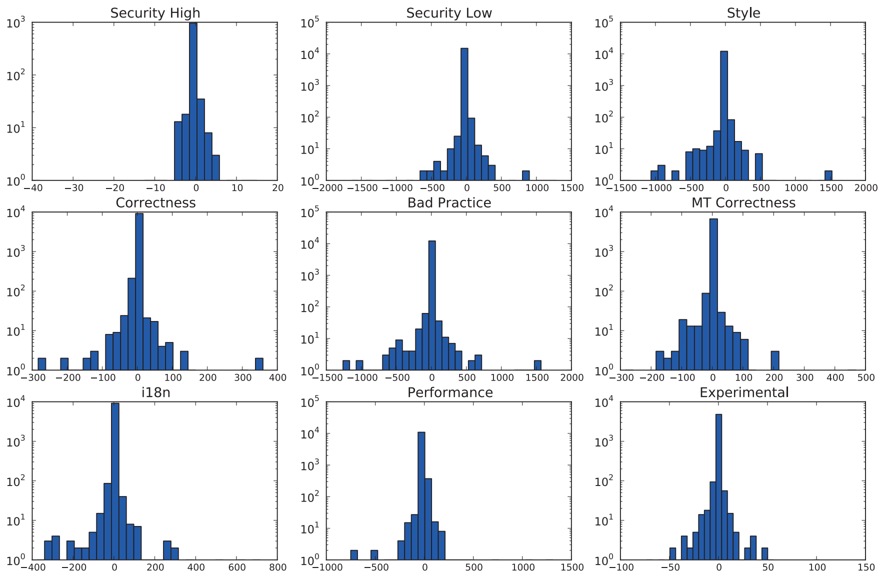

Figure 5: Changes in bug counts between versions.

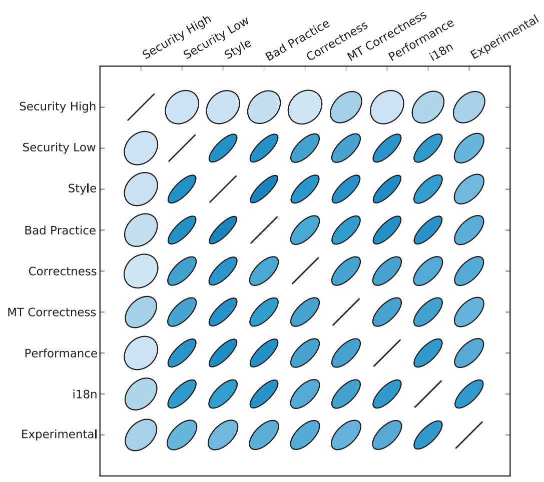

Figure 6: Correlation matrix plot for bug categories.

*High* category showed the weakest correlations with the other categories. Our results show that in general **bugs do flock together**. We do not find projects with only a certain kind of bug; bugs come upon projects in swarms of different kinds. Bugs of the *Security High* category, though, are different: they are either not associated with other bugs, or only weakly so. Perhaps it takes a special kind of blunder to make it a security hazard. Thus, to find such defects, code reviewers with experience in software security issues might be needed.

## 5 Conclusions and Future Work

We analysed more than 260GB of interdependent project versions to see how security bugs evolve over time, their persistence, their relation with other bug categories, and their relationship with size in terms of bytecode.[6]

Our primary hypothesis was that security bugs, and especially severe ones, would be corrected as projects evolve. We found that, although bugs do close over time in particular projects, we do not have an indication that across projects they decrease as projects mature. Moreover, defect counts may increase, as well as decrease in time. Contrary to our second research hypothesis, we found that security bugs are not eliminated in a way that is particularly different from the other bugs. Also, having an average of two to three versions persistence in a sample where 60% of the projects have three versions, is not a positive result especially in the case of the *Security High* bugs. Concerning the relation between severe security bugs and a project's size we showed that they are not proportionally related. Given that, we could say that it would be productive to search for and fix security bugs even if a project grows bigger. Furthermore, the pairwise correlations between all categories indicated that even though all the other categories are related, severe bugs do not appear together with the other bugs. Also, it is interesting to see that security bugs were one of the top two bug categories existing in a large ecosystem. Finally, we highlighted the domino effect, and showed evidence that indicates that Linus's Law does not apply in the case of the security bugs.

Contrary to the approaches that examine versions formed after every change that has been committed to the repository, our observations are made from a different perspective. The versions examined in this work were actual releases of the projects. As a result we do not have an indication of how many changes have been made between the releases. In essence, these JARs were the ones that were or still are, out there in the wild, being used either as applications, or dependencies of others.

Furthermore, the fact that projects have their own idiosyncrasies concerning security bugs, could help us answer questions like: what are the common characteristics of the projects where security bugs increase over time? In addition, by examining source code repositories more closely we could see how different development styles (i.e. size of commits, number of developers) affect projects.

By selecting an large ecosystem that includes applications written only in Java, we excluded by default measurements that involve vulnerabilities like the infamous buffer overflow defects [21]. Still, by examining software artifacts with similar characteristics facilitates the formation of an experiment. Thus, future work on our approach could also involve the observation of other ecosystems, that serve different languages, in the same manner such as, Python's PyPY (Python Package Index), Perl's CPAN (Comprehensive Perl Archive Network), and Ruby's RubyGems.

## 6 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 - Research Funding Program: Heracleitus II. Investing in knowledge society through the ESF.

## References

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

[2] BACA, D., CARLSSON, B., AND LUNDBERG, L. Evaluating the cost reduction of static code analysis for software security. In

*Proceedings of the third ACM SIGPLAN workshop on Programming languages and analysis for security* (New York, NY, USA, 2008), PLAS '08, ACM, pp. 79–88.

[3] BHATTACHARYA, P. Using software evolution history to facilitate development and maintenance. In *Proceedings of the 33rd International Conference on Software Engineering* (New York, NY, USA, 2011), ICSE '11, ACM, pp. 1122–1123.

[4] BHATTACHARYA, P., AND NEAMTIU, I. Bug-fix time prediction models: can we do better? In *Proceedings of the 8th Working Conference on Mining Software Repositories* (New York, NY, USA, 2011), MSR '11, ACM, pp. 207–210.

[5] BRIN, S., AND PAGE, L. The anatomy of a large-scale hypertextual web search engine. *Comput. Netw. ISDN Syst. 30*, 1-7 (Apr. 1998), 107–117.

[6] CHESS, B., AND WEST, J. *Secure programming with static analysis*, first ed. Addison-Wesley Professional, 2007.

[7] D'AMBROS, M. Supporting software evolution analysis with historical dependencies and defect information. In *ICSM* (2008), pp. 412–415.

[8] D'AMBROS, M., AND LANZA, M. A flexible framework to support collaborative software evolution analysis. In *CSMR* (2008), pp. 3–12.

[9] EDWARDS, N., AND CHEN, L. An historical examination of open source releases and their vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security* (New York, NY, USA, 2012), CCS '12, ACM, pp. 183–194.

[10] FISCHER, M., PINZGER, M., AND GALL, H. Populating a release history database from version control and bug tracking systems. In *Proceedings of the International Conference on Software Maintenance* (Washington, DC, USA, 2003), ICSM '03, IEEE Computer Society, pp. 23–.

[11] GIL, J. Y., AND MAMAN, I. Micro patterns in java code. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 2005), OOPSLA '05, ACM, pp. 97–116.

[12] GÖRG, C., AND WEISSGERBER, P. Error detection by refactoring reconstruction. In *Proceedings of the 2005 international workshop on Mining software repositories* (New York, NY, USA, 2005), MSR '05, ACM, pp. 1–5.

[13] GRAVES, T. L., KARR, A. F., MARRON, J. S., AND SIY, H. Predicting fault incidence using software change history. *IEEE Trans. Softw. Eng. 26*, 7 (2000), 653–661.

[14] HECKMAN, S., AND WILLIAMS, L. On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement* (New York, NY, USA, 2008), ESEM '08, ACM, pp. 41–50.

[15] HECKMAN, S., AND WILLIAMS, L. A model building process for identifying actionable static analysis alerts. In *Proceedings of the 2009 International Conference on Software Testing Verification and Validation* (Washington, DC, USA, 2009), ICST '09, IEEE Computer Society, pp. 161–170.

[16] HETTMANSPERGER, T. P., AND MCKEAN, J. W. *Robust nonparametric statistical methods*. Kendall's Library of Statistics, 1998.

[17] HINDLE, A., AND GERMAN, D. M. SCQL: a formal model and a query language for source control repositories. In *Proceedings of the 2005 international workshop on Mining software repositories* (New York, NY, USA, 2005), MSR '05, ACM, pp. 1–5.

[18] HOVEMEYER, D., AND PUGH, W. Finding bugs is easy. *SIGPLAN Not. 39*, 12 (Dec. 2004), 92–106.

[19] HOVEMEYER, D., AND PUGH, W. Finding more null pointer bugs, but not too many. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering* (New York, NY, USA, 2007), PASTE '07, ACM, pp. 9–14.

[20] KAGDI, H., COLLARD, M. L., AND MALETIC, J. I. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *J. Softw. Maint. Evol. 19*, 2 (Mar. 2007), 77–131.

[21] KEROMYTIS, A. D. Buffer overflow attacks. In *Encyclopedia of Cryptography and Security (2nd Ed.)*. 2011, pp. 174–177.

[22] KIM, M., CAI, D., AND KIM, S. An empirical investigation into the role of api-level refactorings during software evolution. In *Proceedings of the 33rd International Conference on Software Engineering* (New York, NY, USA, 2011), ICSE '11, ACM, pp. 151–160.

[23] KIM, S., AND ERNST, M. D. Which warnings should i fix first? In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering* (New York, NY, USA, 2007), ESEC-FSE '07, ACM, pp. 45–54.

[24] KIM, S., PAN, K., AND WHITEHEAD, JR., E. J. Micro pattern evolution. In *Proceedings of the 2006 international workshop on Mining software repositories* (New York, NY, USA, 2006), MSR '06, ACM, pp. 40–46.

[25] LANYUE LU, ANDREA C. ARPACI-DUSSEAU, REMZI H. ARPACI-DUSSEAU, SHAN LU. A Study of Linux File System Evolution. In *Proceedings of the 11th Conference on File and Storage Technologies (FAST '13)* (San Jose, California, February 2013).

[26] LEHMAN, M. M., RAMIL, J. F., WERNICK, P. D., PERRY, D. E., AND TURSKI, W. M. Metrics and laws of software evolution - the nineties view. In *Proceedings of the 4th International Symposium on Software Metrics* (Washington, DC, USA, 1997), METRICS '97, IEEE Computer Society, pp. 20–.

[27] MASSACCI, F., NEUHAUS, S., AND NGUYEN, V. H. After-life vulnerabilities: a study on firefox evolution, its vulnerabilities, and fixes. In *Proceedings of the Third international conference on Engineering secure software and systems* (Berlin, Heidelberg, 2011), ESSoS'11, Springer-Verlag, pp. 195–208.

[28] NAGAPPAN, N., BALL, T., AND ZELLER, A. Mining metrics to predict component failures. In *ICSE '06: Proceedings of the 28th international conference on Software engineering* (New York, NY, USA, 2006), ACM, pp. 452–461.

[29] NUSSBAUM, L., AND ZACCHIROLI, S. The ultimate debian database: Consolidating bazaar metadata for quality assurance and data mining. In *Proceedings of the 2010 international workshop on Mining software repositories* (2010), MSR '10, pp. 52–61.

[30] OZMENT, A., AND SCHECHTER, S. E. Milk or wine: does software security improve with age? In *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15* (Berkeley, CA, USA, 2006), USENIX-SS'06, USENIX Association.

[31] RATZINGER, J., SIGMUND, T., AND GALL, H. C. On the relation of refactorings and software defect prediction. In *Proceedings of the 2008 international working conference on Mining software repositories* (New York, NY, USA, 2008), MSR '08, ACM, pp. 35–38.

[32] RAY, D., AND LIGATTI, J. Defining code-injection attacks. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2012), POPL '12, ACM, pp. 179–190.

[33] SHAHRIAR, H., AND ZULKERNINE, M. Mitigating program security vulnerabilities: Approaches and challenges. *ACM Comput. Surv. 44*, 3 (June 2012), 11:1–11:46.

[34] SHAHZAD, M., SHAFIQ, M. Z., AND LIU, A. X. A large scale exploratory analysis of software vulnerability life cycles. In *Proceedings of the 2012 International Conference on Software Engineering* (Piscataway, NJ, USA, 2012), ICSE 2012, IEEE Press, pp. 771–781.

[35] SHEN, V. Y., YU, T.-J., THEBAUT, S. M., AND PAULSEN, L. R. Identifying error-prone software an empirical study. *IEEE Trans. Softw. Eng. 11*, 4 (Apr. 1985), 317–324.

[36] SPACCO, J., HOVEMEYER, D., AND PUGH, W. Tracking defect warnings across versions. In *Proceedings of the 2006 international workshop on Mining software repositories* (New York, NY, USA, 2006), MSR '06, ACM, pp. 133–136.

[37] STAMAT, M. L., AND HUMPHRIES, J. W. Training $\neq$ education: putting secure software engineering back in the classroom. In *Proceedings of the 14th Western Canadian Conference on Computing Education* (New York, NY, USA, 2009), WCCCE '09, ACM, pp. 116–123.

[38] TELANG, R., AND WATTAL, S. Impact of software vulnerability announcements on the market value of software vendors - an empirical investigation. In *Workshop on the Economics of Information Security* (2007), p. 677427.

[39] TEVIS, J.-E. J., AND HAMILTON, J. A. Methods for the prevention, detection and removal of software security vulnerabilities. In *Proceedings of the 42nd annual Southeast regional conference* (New York, NY, USA, 2004), ACM-SE 42, ACM, pp. 197–202.

[40] WEISSGERBER, P., AND DIEHL, S. Are refactorings less error-prone than other changes? In *Proceedings of the 2006 international workshop on Mining software repositories* (New York, NY, USA, 2006), MSR '06, ACM, pp. 112–118.

[41] ZAMAN, S., ADAMS, B., AND HASSAN, A. E. Security versus performance bugs: a case study on firefox. In *Proceedings of the 8th Working Conference on Mining Software Repositories* (New York, NY, USA, 2011), MSR '11, ACM, pp. 93–102.

[42] ZIMMERMANN, T., NAGAPPAN, N., AND ZELLER, A. Predicting bugs from history. In *Software Evolution*. 2008, pp. 69–88.

## Notes

[1]`http://findbugs.sourceforge.net/`

[2]`http://mvnrepository.com/`

[3]`http://www.mozilla.org/projects/security/known-vulnerabilities.html`

[4]`http://findbugs.sourceforge.net/bugDescriptions.html`

[5]`http://cwe.mitre.org/top25/index.html`

[6]Our data and code are available online at: `https://github.com/bkarak/evol_security_publication_2012`

Table 6: Bug persistence comparison.

| | Security Low | Style | Correctness | Bad Practice | MT Correctness | i18n | Performance | Experimental |
|---|---|---|---|---|---|---|---|---|
| Security High | *(0.04, $p = 0.97$<br>2.72, 2.36<br>243, 35048)* | 2.22, $p < 0.05$<br>2.72, 2.12<br>243, 49043 | *(−0.51, $p = 0.61$<br>2.72, 2.50<br>243, 12905)* | 2.77, $p < 0.01$<br>2.72, 2.11<br>243, 49324 | *(1.02, $p = 0.31$<br>2.72, 2.48<br>243, 10227)* | *(−1.19, $p = 0.23$<br>2.72, 2.74<br>243, 10718)* | *(−1.00, $p = 0.32$<br>2.72, 2.65<br>243, 23598)* | *(−0.33, $p = 0.74$<br>2.72, 2.85<br>243, 2686)* |
| Security Low | | 20.27, $p \ll 0.05$<br>2.36, 2.12<br>35048, 49043 | −3.59, $p \ll 0.05$<br>2.36, 2.50<br>35048, 12905 | 25.17, $p \ll 0.05$<br>2.36, 2.11<br>35048, 49324 | 5.59, $p \ll 0.05$<br>2.36, 2.48<br>35048, 10227 | −7.55, $p \ll 0.05$<br>2.36, 2.74<br>35048, 10718 | −8.19, $p \ll 0.05$<br>2.36, 2.65<br>35048, 23598 | *(−1.39, $p = 0.17$<br>2.36, 2.85<br>35048, 2686)* |
| Style | | | −17.96,<br>$p \ll 0.05$<br>2.12, 2.50<br>49043, 12905 | 5.66, $p \ll 0.05$<br>2.12, 2.11<br>49043, 49324 | −6.84, $p \ll 0.05$<br>2.12, 2.48<br>49043, 10227 | −20.61,<br>$p \ll 0.05$<br>2.12, 2.74<br>49043, 10718 | −26.18,<br>$p \ll 0.05$<br>2.12, 2.65<br>49043, 23598 | −8.30, $p \ll 0.05$<br>2.12, 2.85<br>49043, 2686 |
| Correctness | | | | 21.38, $p \ll 0.05$<br>2.50, 2.11<br>12905, 49324 | 7.44, $p \ll 0.05$<br>2.50, 2.48<br>12905, 10227 | −3.57, $p \ll 0.05$<br>2.50, 2.74<br>12905, 10718 | −2.91, $p < 0.01$<br>2.50, 2.65<br>12905, 23598 | *(0.40, $p = 0.69$<br>2.50, 2.85<br>12905, 2686)* |
| Bad Practice | | | | | −10.02,<br>$p \ll 0.05$<br>2.11, 2.48<br>49324, 10227 | −23.63,<br>$p \ll 0.05$<br>2.11, 2.74<br>49324, 10718 | −30.32,<br>$p \ll 0.05$<br>2.11, 2.65<br>49324, 23598 | −9.98, $p \ll 0.05$<br>2.11, 2.85<br>49324, 2686 |
| MT Correctness | | | | | | −10.17,<br>$p \ll 0.05$<br>2.48, 2.74<br>10227, 10718 | −10.83,<br>$p \ll 0.05$<br>2.48, 2.65<br>10227, 23598 | −4.03, $p \ll 0.05$<br>2.48, 2.85<br>10227, 2686 |
| i18n | | | | | | | *(1.29, $p = 0.20$<br>2.74, 2.65<br>10718, 23598)* | 2.46, $p < 0.05$<br>2.74, 2.85<br>10718, 2686 |
| Performance | | | | | | | | *(1.92, $p = 0.05$<br>2.65, 2.85<br>23598, 2686)* |

The matrix presents pairwise Mann-Whitney $U$ test results between the different bug categories. Each cell contains the test result (the value of $U$), the $p$-value, the average for each category and the sample size for each category. Cells in brackets show pairs where no statistically significant difference was found.