The Vulnerability Dataset of a Large Software Ecosystem

Dimitris Mitropoulos,[*] Georgios Gousios,[†] Panagiotis Papadopoulos,[‡]
Vassilios Karakoidas,[*] Panos Louridas[*] and Diomidis Spinellis[*]
[*]Department of Management Science and Technology
Athens University of Economics and Business
{dimitro, bkarak, louridas, dds}@aueb.gr
[†] Software Engineering Research Group
Delft University of Technology
G.Gousios@tudelft.nl
[‡] Institute of Computer Science
FORTH
panpap@ics.forth.gr

***Abstract*—Security bugs are critical programming errors that can lead to serious vulnerabilities in software. Examining their behaviour and characteristics within a software ecosystem can provide the research community with data regarding their evolution, persistence and others. We present a dataset that we produced by applying static analysis to the Maven Central Repository (approximately 265GB of data) in order to detect potential security 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 in our data repository, we also store specific metadata, such as the JAR's size, its dependencies and others. Our dataset can be used to produce interesting research results involving security bugs, as we show in specific examples.**

***Index Terms*—Security Bugs, Software Security, Static Analysis, FindBugs, Software Ecosystem, Maven Repository, Software Evolution.**

I. Introduction

A security bug is a programming error that introduces a potentially exploitable weakness into a computer system [1]. Compared to other bug categories, failures due to security bugs have two distinct features: they can severely affect an organization's infrastructure [2], and they can cause significant financial damage to an organization [3], [4]. 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. 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 [5]. 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.

TABLE I
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 software ecosystem can be seen as a collection of software projects, which are developed and co-evolved in the same environment [7]. 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 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 that can depend on

[1] http://pypy.org/
[2] http://www.cpan.org/
[3] http://rubygems.org/
[4] http://mvnrepository.com/

TABLE II
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

other versions.[5]

To analyze the Maven repository we used FindBugs,[6] a static analysis tool that was also used for research purposes in [8] and [9]. FindBugs' role is to examine Java bytecode to detect software bugs and separate them into nine categories. Two of them involve security issues (see Table I). 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 concerning security bugs.

II. CONSTRUCTION PROCESS

Initially, we obtained a snapshot of the Maven repository and handled it locally to retrieve a list of all the names of the project versions that existed in it. Then, we filtered out projects written in programming languages other than Java because FindBugs analyzes only Java bytecode. The statistic measurements concerning the repository can be seen in Table II.

Due to the large volume of our dataset, we designed our data processing step in a distributed way. A schematic representation of our data processing architecture can be seen in Figure 1. In particular, we created a series of processing tasks based on the JAR list we have obtained and added them to a task queue mechanism (a RabbitMQ[7] 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[8] database system).

A typical processing cycle of a worker included the following steps: as soon as the worker was 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 III). 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.

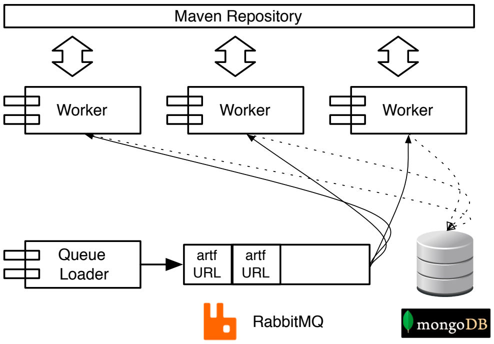


Fig. 1. The data processing architecture.

TABLE III
BUG DESCRIPTION.

`type`	EI_EXPOSE_REP2
`category`	MALICIOUS_CODE
`source File`	`ANTLRHashString.java`
`class`	`antlr.ANTLRHashString`
`method`	`setBuffer`
`sourceline start`	97
`sourceline end`	98

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).

III. DATASET ENTRIES

FindBugs 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 (see Table III). FindBugs' results also include additional information like the number of classes included in the examined JAR and others.

As we mentioned earlier, our data were stored in a MongoDB database that stores its records in JSON-like documents. However, FindBugs 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 discussed in Section II, 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, and the ordinal version number of the version. 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 metadata each worker collected. Note that the results of FindBugs are too large to fit, thus in order to see a complete instance please visit our GitHub repository (see Section VII):

[5] Note that in the Maven repository, versions are *actual* releases.
[6] http://findbugs.sourceforge.net/
[7] http://www.rabbitmq.com/
[8] http://www.mongodb.org/

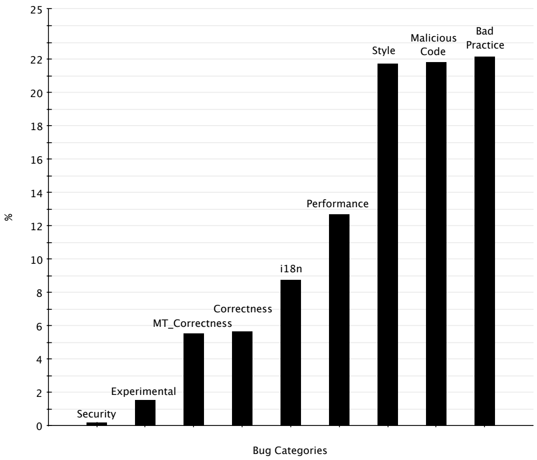


Fig. 2. Bug percentage in Maven repository.

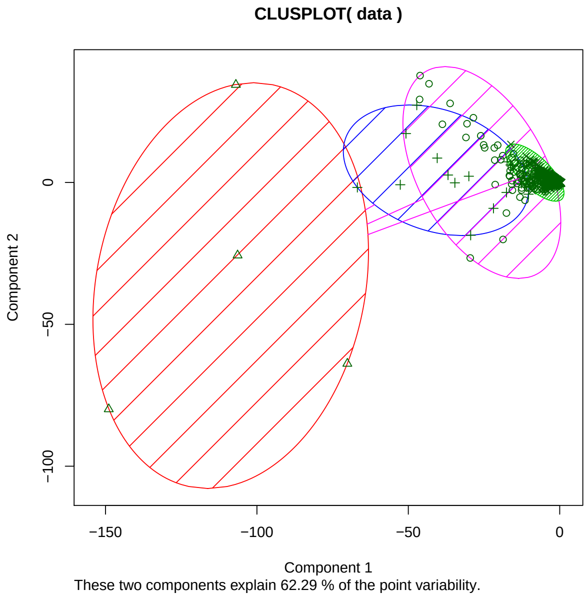


Fig. 3. Basic *k-means* clusters of all the versions that exist in the ecosystem.

```
{"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 */ }
}
```

TABLE IV
CORRELATIONS BETWEEN VERSION AND SOFTWARE BUGS 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	$\ll 0.05$

IV. GATHERING EXPERIENCE RETURNS FOR SECURITY BASED ON OUR DATASET

Since MongoDB provides a rich query interface, it was easy to find out how software bugs are distributed among the repository (see Figure 2) or identify the main clusters that are formed based on the number of the bugs of every version (see Figure 3). An interesting observation is that the *Malicious Code* bugs, together with the *Bad Practice* bugs are the most popular in the repository. Also, a simple query like the following, will reveal that from the total number of versions, 45,559 of them contained at least one bug coming from the *Malicious Code* category:

```
db.findbugs.find({
     'BugCollection.BugInstance.category' :
     'MALICIOUS_CODE'}).count()
```

Another observation involves specific bugs thath we could consider as critical 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 [10]. Also, as FindBugs' bug descriptions indicate,[9] if an application has bugs coming from this category, it might have more vulnerabilities that FindBugs doesn't report. Table V presents the number of versions where at least one of these bugs exists. In essence, 5,341 project versions, contained at least one bug related to user-input validation issues. 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 security bugs.

Furthermore, we have created a series of scripts to exhibit how the dataset can be used to capture correlations regarding the evolution of security 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 IV). 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 IV applies to all versions of all projects together.

[9] http://findbugs.sourceforge.net/bugDescriptions.html

TABLE V

NUMBER OF PROJECT VERSIONS THAT CONTAIN AT LEAST ONE SECURITY BUG RELATED TO USER-INPUT VALIDATION ISSUES.

Bug Description	Number of Project Versions
HRS: HTTP cookie formed from untrusted input	151
HRS: HTTP response splitting vulnerability	1,579
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

TABLE VI

CORRELATIONS BETWEEN JAR SIZE AND SOFTWARE BUGS 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	$\ll 0.05$

The situation was different in individual projects where we performed Spearman correlations between security 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. Such results indicate that we cannot say if vulnerabilities decrease or increase as projects mature.

In addition, we explored the relation between security bugs 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 security bug counts for each project and version. The results can be seen in Table VI. An interesting observation is that the *Security* category stands out by having a remarkably lower effect than the other categories. As we mentioned earlier, many bugs that belong to this category are related to user-input validation issues. Hence, it seems that 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.

Figure 5 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. Our results show that bugs coming from the *Security* category are not correlated with the bugs coming from other categories. This indicates that security bugs of this kind do not appear together with the other bugs.[10]

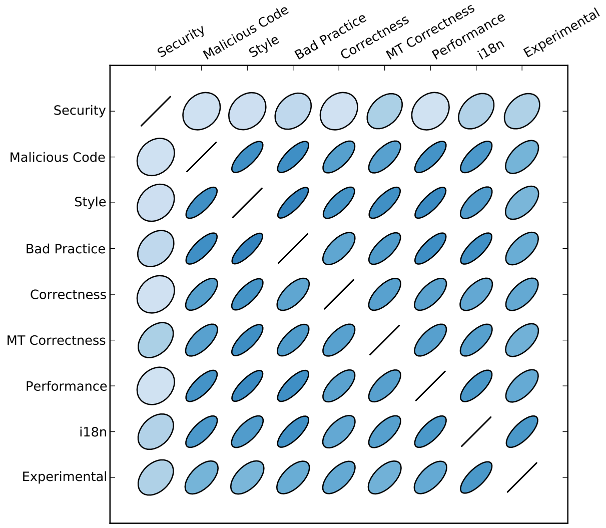


Fig. 5. Correlation matrix plot for bug categories. Blue color indicates positive correlation. The darker the color (and the more acute the ellipsis slant), the stronger the correlation.

V. THREATS TO VALIDITY

During our dataset analysis we faced some limitations that concerned the non-availability of some JARs. Specifically, there were some JARs included in the initial JAR list, that were not available online, when the FindBugs result collection was performed.

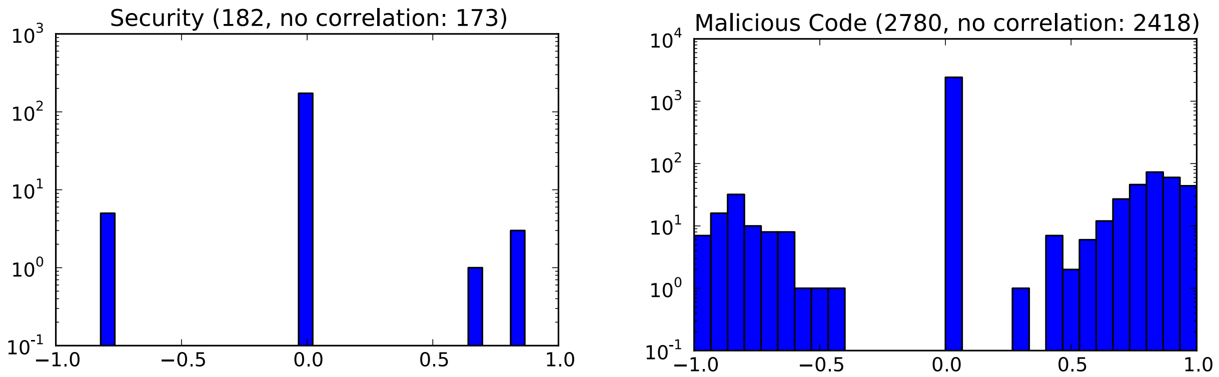
A threat to the internal validity of our dataset construction process could be the false alarms of the FindBugs tool [8], [12]. Specifically, reported security bugs may not be applicable to an application's typical use context. For instance, FindBugs could report an SQL injection vulnerability in an application that receives no external input. In this particular context, this would be a false positive alarm. False alarms of static analysis tools and how they can be reduced are issues that have already been discussed in literature [9], [13] and they are beyond the scope of this paper.

VI. RELATED WORK

Our work is related to the creation of datasets to facilitate research and the examination of software vulnerabilities.

The Maven ecosystem has been previously analyzed by Raemaekers et al. [14] to produce the *Maven dependency dataset*. Apart from basic information like individual methods,

[10] Further research concerning the examination of security bugs based on this dataset can be found in our previous paper [11].



(a)

(b)

Fig. 4. Correlations between version and software bugs count.

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.

On the security front, Ozment and Schechter [15] examined the code base of the OpenBSD operating system to determine whether its security is increasing over time. Massacci et al. [16] observed the evolution of software defects by examining six major versions of Firefox. In addition, Shahzad et al. [1] analysed large sets of vulnerability data-sets to observe various features of the vulnerabilities that they considered critical, while Edwards et al. [17] have examined the evolution of security bugs by examining different versions of four projects.

VII. Conclusions

In this paper, we have presented a dataset that includes all the software bugs that each JAR of the Maven central repository contains along with some other metadata mentioned in Section IV. We have also shown how our data can be used to extract results concerning the evolution and the behaviour of security bugs.

Initially, we made some observations concerning the security bugs of the Maven repository as a whole. Then, based on our dataset, we found that we cannot say with confidence if security bugs increase or decrease as projects mature. We also showed that there were many projects where security bug counts do not change as projects evolve. 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 is productive to search for and fix security bugs even if a project grows bigger. In addition, the pairwise correlations between all categories indicates that even though all the other categories are related, severe bugs do not appear together with the other bugs. Such findings indicate that projects have their own idiosyncrasies regarding security bugs and could help us answer questions like: what are the common characteristics of the projects where security bugs increase over time?

By selecting a large ecosystem that includes applications written only in Java, we excluded by default measurements that involve vulnerabilities like the infamous buffer overflow vulnerabilities [18]. 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 like the ones mentioned in Section I and projects in different languages like Ruby, Python *etc.* Concluding, the complete set of our data and source code will become available upon publication.

VIII. Acknowledgments

The project is being co-financed by the European Regional Development Fund (ERDF) and national funds and is a part of the Operational Programme "Competitiveness & Entrepreneurship" (OPCE II), Measure "COOPERATION" (Action I).

References

[1] M. Shahzad, M. Z. Shafiq, and A. X. Liu, "A large scale exploratory analysis of software vulnerability life cycles," in *Proceedings of the 2012 International Conference on Software Engineering*, ser. ICSE 2012. Piscataway, NJ, USA: IEEE Press, 2012, pp. 771–781.

[2] H. Shahriar and M. Zulkernine, "Mitigating program security vulnerabilities: Approaches and challenges," *ACM Comput. Surv.*, vol. 44, no. 3, pp. 11:1–11:46, Jun. 2012.

[3] J.-E. J. Tevis and J. A. Hamilton, "Methods for the prevention, detection and removal of software security vulnerabilities," in *Proceedings of the 42nd annual Southeast regional conference*, ser. ACM-SE 42. New York, NY, USA: ACM, 2004, pp. 197–202. [Online]. Available: http://doi.acm.org/10.1145/986537.986583

[4] D. Baca, B. Carlsson, and L. Lundberg, "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*, ser. PLAS '08. New York, NY, USA: ACM, 2008, pp. 79–88.

[5] R. Telang and S. Wattal, "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.

[6] B. Chess and J. West, *Secure programming with static analysis*, 1st ed. Addison-Wesley Professional, 2007.

[7] 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*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 289–292. [Online]. Available: http://doi.acm.org/10.1145/1810295.1810356

[8] N. Ayewah and W. Pugh, “The google FindBugs fixit,” in *Proceedings of the 19th international symposium on Software testing and analysis*, ser. ISSTA ’10. New York, NY, USA: ACM, 2010, pp. 241–252.

[9] J. Spacco, D. Hovemeyer, and W. Pugh, “Tracking defect warnings across versions,” in *Proceedings of the 2006 international workshop on Mining software repositories*, ser. MSR ’06. New York, NY, USA: ACM, 2006, pp. 133–136.

[10] D. Ray and J. Ligatti, “Defining code-injection attacks,” in *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL ’12. New York, NY, USA: ACM, 2012, pp. 179–190. [Online]. Available: http://doi.acm.org/10.1145/2103656.2103678

[11] 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*. Usenix Association, Oct. 2013, pp. 37–48. [Online]. Available: http://www.dmst.aueb.gr/dds/pubs/conf/2013-LASER-BugEvol/docs/html/evol.html

[12] D. Hovemeyer and W. Pugh, “Finding bugs is easy,” *SIGPLAN Not.*, vol. 39, no. 12, pp. 92–106, Dec. 2004.

[13] S. Heckman and L. Williams, “A model building process for identifying actionable static analysis alerts,” in *Proceedings of the 2009 International Conference on Software Testing Verification and Validation*, ser. ICST ’09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 161–170.

[14] 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*, ser. MSR ’13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 221–224. [Online]. Available: http://dl.acm.org/citation.cfm?id=2487085.2487129

[15] A. Ozment and S. E. Schechter, “Milk or wine: does software security improve with age?” in *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15*, ser. USENIX-SS’06. Berkeley, CA, USA: USENIX Association, 2006.

[16] F. Massacci, S. Neuhaus, and V. H. Nguyen, “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*, ser. ESSoS’11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 195–208.

[17] N. Edwards and L. Chen, “An historical examination of open source releases and their vulnerabilities,” in *Proceedings of the 2012 ACM conference on Computer and communications security*, ser. CCS ’12. New York, NY, USA: ACM, 2012, pp. 183–194. [Online]. Available: http://doi.acm.org/10.1145/2382196.2382218

[18] A. D. Keromytis, “Buffer overflow attacks,” in *Encyclopedia of Cryptography and Security (2nd Ed.)*, 2011, pp. 174–177.