Fixing of Security Vulnerabilities in Open Source Projects: A Case Study of Apache HTTP Server and Apache Tomcat

Valentina Piantadosi
University of Molise, Italy
v.piantadosi@studenti.unimol.it

Simone Scalabrino
University of Molise, Italy
simone.scalabrino@unimol.it

Rocco Oliveto
University of Molise, Italy
rocco.oliveto@unimol.it

***Abstract*—Software vulnerabilities are particularly dangerous bugs that may allow an attacker to violate the confidentiality, integrity or availability constraints of a software system. Fixing vulnerabilities *soon* is of primary importance; besides, it is crucial to release *complete* patches that do not leave any corner case not covered. In this paper we study the process of vulnerability fixing in Open Source Software. We focus on three dimensions: *personal*, *i.e.*, who fixes software vulnerabilities; *temporal*, *i.e.*, how long does it take to release a patch; *procedural*, *i.e.*, what is the process followed to fix the vulnerability. In the context of our study we analyzed 337 CVE Entries regarding Apache HTTP Server and Apache Tomcat and we manually linked them to the patches written to fix such vulnerabilities and their related commits. The results show that developers who fix software vulnerabilities are much more experienced than the average. Furthermore, we observed that the vulnerabilities are fixed through more than a commit and, surprisingly, that in about 3% of the cases such vulnerabilities show up again in future releases (*i.e.*, they are not actually fixed). In the light of such results, we derived some lessons learned that represent a starting point for future research directions aiming at better supporting developers during the documentation and fixing of vulnerabilities.**

***Index Terms*—Software Vulnerabilities, Empirical Studies, Mining Software Repositories, Open-Source Software**

I. Introduction

Software vulnerabilities are a big threat for the security of software systems. Vulnerabilities are bugs, *i.e.*, errors in the source code, that can be exploited to (i) take control of the system (*i.e.*, *integrity*), (ii) acquire private data (*i.e.*, *confidentiality*), and (iii) take the system down (*i.e.*, *availability*).

Software vulnerabilities are not only dangerous for the users. They are critical also for the companies that maintain the software affected and the ones that use such software. As reported in a previous study, software vulnerabilities have a significant impact on the stock price of the companies when they are disclosed and exploited [20].

During software maintenance, vulnerabilities should be addressed with higher priority, since they can harm the users in a more severe way compared to normal bugs. Besides, because of the hazard entailed by using vulnerable software in production, such bugs should be disclosed only when they are properly patched and, ideally, deployed. For this reason, organizations usually encourage the users who find vulnerabilities communicating them privately, some of them even rewarding people who do that[1].

Then, an internal team of security experts works on the patch to minimize the probability that an attacker exploits them on production systems, and, as soon as a new version is released, the vulnerability is publicly disclosed and users can deploy the patched version.

This can happen in several ways. Some projects, such the ones in the Apache ecosystem, provide newsletters about security updates available. Moreover, the NIST (National Institute for Standards and Technology) and MITRE maintain CVE (Common Vulnerabilities and Exposure) databases, *i.e.*, dictionaries that provide information about publicly disclosed vulnerabilities and exposures. Each vulnerability has a CVE Entry, and each entry has a unique ID, a description and an estimation of the severity of the vulnerability. The main goal of CVE is to provide a way to share information about vulnerabilities among different tools.

Despite the presence of such sources of information, reconstructing what happens from the report to the release of the patch *a-posteriori* from the CVE Entries can be tricky. CVE Entries, indeed, often do not report information about vulnerable code and patches produced.

Recently, Li and Paxson [10] used a fully-automated approach to conduct a large-scale analysis on vulnerability patching. Such a study provides useful insights about vulnerability patching from a *security* perspective. Besides, Nappa *et al.* [17] studied how quickly vulnerability patches are deployed. However, to the best of our knowledge, no previous study analyzed the characteristics of the *engineering* process behind vulnerability fixing. Indeed, it is still not clear *who* are the developers who address vulnerabilities, *what* is the timing of vulnerability fixing, and *how* the patches are produced. Such an analysis would be useful to understand what is the best way to address vulnerabilities (*e.g.*, assigning them to experienced developers or to the developers who worked mostly on the vulnerable feature).

In this paper we bridge this gap. We conduct an in-depth empirical study to investigate the process of vulnerability fixing in open-source projects.

[1] https://www.google.com/about/appsecurity/reward-program/

We focus on three aspects:

1) *Who are the developers who fix the vulnerabilities?*
2) *When are vulnerabilities fixed?*
3) *How are vulnerabilities fixed?*

To answer the above questions, we considered all the vulnerabilities reported in the CVE database for two open-source projects, *i.e.*, Apache HTTP and Apache Tomcat. We manually linked each vulnerability to the fixing commits to have a reliable source for our analysis. Then, we used such a dataset to answer our research questions.

Our results show that the developers who fix software vulnerabilities are generally much more experienced than the average team members; besides, we found that such developers rarely modify files they own. Software vulnerabilities are usually fixed before the related CVE Entries are published; however, we found cases in which the fix was completed even years after the CVE Entry was published. Finally, we show that most of the vulnerabilities are fixed with very small patches.

We used the results achieved to derive some lessons learned about vulnerability fixing. For example, our results suggest that using a vulnerability patching process more similar to the one used in closed-source software (with a small team handling such cases) seems to provide some benefits in terms of security.

The derived lessons learned call for new approaches supporting developers during vulnerability fixing. For example, we foresee a valuable support from approaches for vulnerability triaging, to assign vulnerabilities to the developer who most likely will be able to fix it effectively and efficiently; also, approaches for detecting incomplete patches could be critical in some contexts to minimize the potential risks of exploit.

The reminder of our paper is organized as follows. In Section II we present the empirical study design of our investigation; Section III contains the detailed results of our analysis; in Section IV we discuss our findings and we report some lessons learned; in Section V we present the threats that could affect the validity of our study, and we conclude our work in Section VII, presenting possible new research directions based on the results achieved in our study.

II. Empirical Study Design

The *goal* of the study is to analyze the process of vulnerability fixing in open source projects looking at three dimensions: (i) *personal* (who fixed the vulnerabilities?); (ii) *temporal* (when are vulnerabilities fixed?); (iii) *procedural* (how are the vulnerabilities fixed?).

Thus, our study is steered by the following research questions:

- RQ$_1$: *Who fixes vulnerabilities?* This research question aims at understanding who are the developers who fix vulnerabilities in a team. We want to check if the developers who fix vulnerabilities are more experienced than the average (i) on the whole project and (ii) on the files they modify.
- RQ$_2$: *When are vulnerabilities fixed?* This research question aims at discovering how much time is needed to fix vulnerabilities and how frequently vulnerabilities show up again after they are (apparently) fixed in a software release.
- RQ$_3$: *How are vulnerabilities fixed?* With this research question we want to analyze the process of vulnerability fixing, looking at the number of commits, at the files involved, and checking if the change involves one or more subsystems (*i.e.*, if they are *local* or *global* changes).

TABLE I: CVE Entries in the Apache ecosystem. In the context of our study we selected the two projects with the highest number of reported software vulnerabilities (reported in bold).

Project	CVE Entries
Apache HTTP Server	**206**
Apache Tomcat	**131**
Apache Struts	71
Apache Subversion	39
Apache ActiveMQ	25

A. Context Selection

The context of this study consists of (i) two open source projects belonging to the Apache ecosystem, *i.e.*, HTTP Server and Tomcat; (ii) their whole history of publicly known vulnerabilities; and (iii) their change history.

We selected HTTP Server and Tomcat from the Apache ecosystem because they are the two projects with the highest number of reported software vulnerabilities. Table I reports the ranking of five software projects from the Apache ecosystem with the largest number of CVE Entries at the time of the experiment.

We relied on CVE Details[2], a mirror of the National Vulnerability Database (NVD), as the data source for software vulnerabilities. We decided to use such a database instead of the NVD or the MITRE CVE database because it provides much useful information about the vulnerabilities. We used such information to track the patching commits. Such a database lists CVE Entries. Each CVE Entry refers to a specific software vulnerability of the project, and it has a unique identifier. The identifiers is assigned by the CVE Numbering Authorities (CNAs).

Each CVE Entry in the CVE Details database contains both structured and unstructured information about the releases of the systems that were affected and the severity of the vulnerability. We collected such information and we enriched it with details about the commits done to patch the vulnerabilities in a fully structured way.

A part of the unstructured information that we had to structure is the list of commits that fixed a vulnerability. Indeed, in the CVE Details database there is not always a direct connection between the CVE Entry and the commit(s) that fixed it. Moreover, some of the patches to the oldest vulnerabilities were posted on the mailing lists and they were

[2] https://cvedetails.com

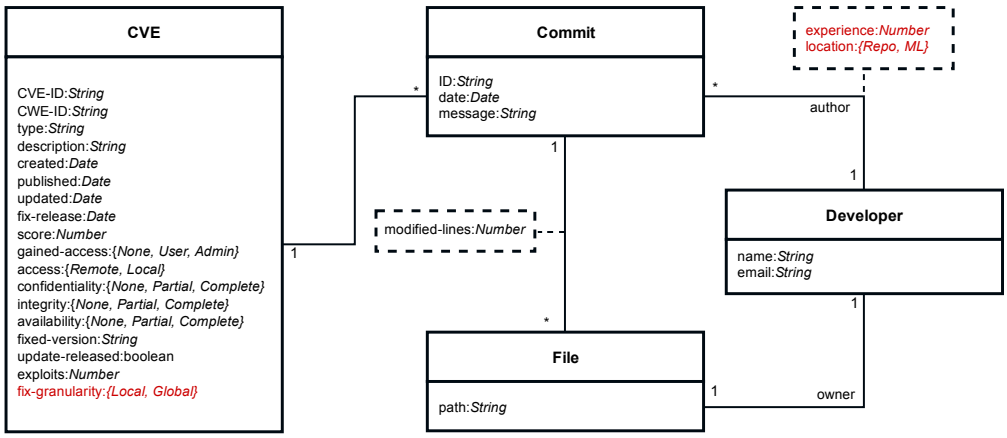

Fig. 1: Conceptual diagram of the dataset we created. In red, the data that we computed to answer our research questions.

committed by other internal members of the project. For this reason, we could not devise an approach to automatically extract such information and we needed to manually analyze each CVE Entry to find the fixing commits and the responsible authors. We explain in the following subsection the process we followed to do this.

In the end, we created a dataset of CVE Entries with (i) details on the vulnerability, (ii) details on patches (*e.g.*, lines of code), (iii) details on the commits involved (*e.g.*, commit time) and (iv) details on the author (*e.g.*, name and username). We use this dataset to answer all our research questions. We report in Figure 1 the conceptual diagram of the dataset we acquired. We report in red the data we computed to answer our research questions, detailed in Section II-C

B. Finding the Fixing Commits

For all the CVE Entries marked as fixed, we had to manually look for the fixing commits in the project repository. To this aim, we used the unstructured information provided in the CVE Entries. Such information contain a list of links to useful resources, such as discussions in the mailing lists.

As previously stated, finding the fixing commits and the authors is not a trivial process. We used a systematic approach to retrieve such information in the most reliable way. As a first step, we tried to find direct and indirect links from the CVE Entry to any GitHub commit and we considered such commits as *potentially fixing*. However, the CVE Entries describing vulnerabilities that involved old releases of Apache HTTP (*e.g.*, before the GitHub repository was created at all) did not contain such links. Also, we found that, even when such a piece of information is available, it is sometimes not complete, *i.e.*, there are other commits related to the vulnerability patching not reported in the CVE Entry.

Therefore, we looked for the fixing commits also in the revision history of the project. First, we considered as *potentially fixing commits* all the commits containing the ID of the CVE Entry in the commit messages in the whole history of the project. Then, we used keywords from the CVE Entry description to find other *potentially fixing commits*. In this case, we focused on the commits from the date in which the earliest vulnerable version of the software was published to date in which the latest patched version was released. Both such dates can be deduced from the data provided by the CVE Entries. Finally, we manually analyzed the set containing all the *potentially fixing commits* we gathered and selected the ones that actually fixed the vulnerability. We summarize in Figure 2 the process we followed to find the fixing commits for a given CVE Entry.

We noticed that some patches for the old vulnerabilities from Apache HTTP were reported also on the mailing list of the project. However, there was a mismatch between the author who committed the patch in the repository and the one who wrote it in the mailing list. We keep both the authors in the dataset separately, because the committers could slightly modify the patch before the commit.

For each commit done to fix the vulnerability, we considered (i) the list of files changed, (ii) the number of lines modified, and (iii) the author of the commit and the name of the author of the patch from the mailing list, if they were different.

We report in Table II the details of our dataset. It is worth noting that some CVE Entries were disputed by the Apache HTTP project members, while in some cases they were not considered as security threats at all. We ignore such vulnerabilities if no commits were done. In one case, we found a disputed CVE Entry with a fixing commit which updated the documentation. We include such a commit in our dataset. Therefore, in total, our dataset contains **239** vulnerabilities with the related fixing commits.

C. Experimental procedure

To answer RQ_1, we first compute the experience of the developers and the ownership of the files using the GitHub

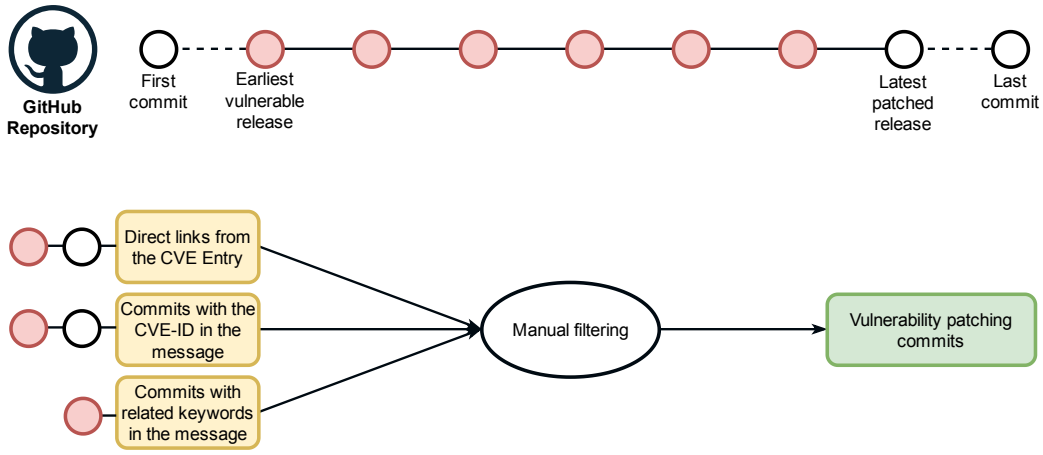

Fig. 2: The process we followed to find fixing commits for a CVE Entry.

TABLE II: Details on the dataset.

Project	CVE Entries					Fixes	
	Fixed	No fix found	Disputed	Not a vulnerability	Duplicated	Commits	Authors
Apache HTTP Server	130	57	3	5	0	378	44
Apache Tomcat	109	16	0	2	1	232	10

repositories of the projects we kept into account. We compute the experience of a developer at a given point in time as the number of commits they did in the repository at that date. Note that the number of commits is in general used as a proxy of developer's experience (see *e.g.*, [18]).

As for the ownership, we first retrieve the commits regarding each file at a given point in time; then, we assign the ownership to the author that did the majority of the changes (*i.e.*, number of added, modified, or deleted lines of code). If many authors did the same number of changes, we assigned the ownership to the one of them who did the earliest commit.

It is worth noting that in both cases we did not compute experience and ownership in general, but in time. Indeed, the projects we considered have a long life time (23 years for Apache HTTP and 19 years for Apache Tomcat) and the experience of an author may significantly vary in time. In this context, we are interested in the experience of an author at the time of the commit. The same is true for the ownership.

We considered commits and patches separately to analyze the experience of the developers who fix vulnerabilities. For all the CVE Entries fixed only with commits, for each vulnerability, we computed (i) the experience of the authors of the fix and (ii) the mean experience of all developers (the team) at the time of the fix. We use a paired Wilcoxon Signed-Rank test to check if the difference between the experience of vulnerability fixers and other members of the team is significant; we reject the null hypothesis (*i.e.*, *there is no difference in terms of experience between vulnerability fixers and other members of the team*) if the *p-value* is lower than 0.05. We also use Cliff's delta [7] to check the magnitude of such a difference. We consider the difference *negligible* if $|\delta| < 0.148$, *small* if $0.148 \leq |\delta| < 0.33$, *medium* if $0.33 \leq |\delta| < 0.474$, and *large* for $|\delta| \geq 0.474$ [8].

We also check if there is a relationship between file ownership and fix to have a better understanding on *who* fixes vulnerabilities. Specifically, we check if the authors of the commits only modify files that they own at the time of the fix. We consider three possible scenarios:

1) all the developers who contribute to the fix only modify only the files they own;
2) none of them modifies the files they own;
3) there are both cases in which they modify files they own and files they do not own.

For each CVE Entry we decide which is the case and we report the frequency of each scenario.

We report the results about the patches found on the mailing lists separately, since we do not have precise data about the experience of the authors. We assume that such authors did not have yet enough experience on the project, therefore they are not allowed to commit directly on the repository. We report the number of external patches that we found and the percentage of authors that became committers of the project later.

To answer RQ_2, we first compute the exposure time of each vulnerability. To do this, we compute the difference between

the date in which the CVE Entry was published and the date in which the last commit regarding the fix of the vulnerability was done. Negative values indicate that the CVE Entry was published *after* the vulnerability was fixed.

It is worth noting that such an estimate is optimistic: we assume that an attacker is aware of a vulnerability only if a CVE Entry is published. However, an attacker can potentially acquire such information from other sources, such as the change log of a release. Therefore, if a vulnerability was not completely fixed after a release i, *i.e.*, it shows up again after in the release $i+1$, an attacker could potentially exploit such a vulnerability. We report the number of vulnerabilities not completely fixed in a release to understand the magnitude of this phenomenon. To this end, we check the versions affected by each vulnerability from its CVE Entry. If two different versions were affected by the same vulnerability, we say that the first fix was *incomplete*, otherwise we assume it was *complete*.

While for Apache HTTP the version naming is straightforward, *e.g.*, version 2.2 comes after version 2.1, for Apache Tomcat the situation is different. Indeed, Tomcat provides parallel major versions. For example, if a vulnerability regards versions 6, 7, and 8, this does not mean that the fix was not complete. Instead, it means that the core part, shared by the three versions, was affected. Therefore, for Tomcat, we say that a CVE Entry had an *incomplete* fix only if at least two different versions of the same branch were affected (*e.g.*, 6.0.1 and 6.0.2).

Finally, we estimate the time needed by the developers to work on the vulnerabilities. To do this, we compute the difference between the dates of the last and the first commit of the fix. It is worth noting that this is a conservative estimation of the actual time needed, since we ignore the time needed to make the first commit. We ignore the fixes that required just a single commit.

To answer RQ_3, for each CVE Entry we compute (i) the number of commits, (ii) the number of files changed, and (iii) the number of lines changed. We report the distributions of such values to understand how a typical vulnerability fix looks like.

Finally, for each CVE, we verify if the change is *global* or *local*. We say that a fix is *local* if it involves only source files belonging to the same subsystem; otherwise, we say that a fix is *global* if it involves source files from many subsystems. In both the cases we assume that the source files in the same folder belong to the same subsystem. We filter source files based on the extensions (*i.e.*, ".h" and ".c" for Apache HTTP and ".java" for Apache Tomcat). Specifically, for Apache Tomcat we consider only the ".java" files in the `java` folder, to ignore all the test cases. We also report the cases in which no source files were modified to fix a vulnerability.

Our research questions and the analysis we did to answer them are summarized in Table III.

TABLE III: Research Questions.

Research Question	Analysis
RQ_1: Who?	Authors' experience Authors' ownership of patched files
RQ_2: When?	Exposure time Completeness of the fix Fix time
RQ_3: How?	Number of commits Patch size Patch locality

D. Replication Package

We release our dataset[3] to insure the replicability of our results and to allow the research community to perform additional analysis. For privacy reasons, we do not report in our dataset the email addresses of the developers.

III. Empirical Study Results

In this section we report the results of our empirical study.

A. RQ_1: Who Fixes Software Vulnerabilities?

In Table IV we compare the mean experience of developers who fix vulnerabilities to the mean experience of the whole developing team. For both the systems, the developers who fix software vulnerabilities are much more experienced than the average.

This difference is particularly evident for Apache Tomcat. In this case, the mean maximum experience of developers over the history of the project is about 6,167 commits, while the mean vulnerability fixers' experience is about 5,295 commits. This means that the developers who fix the vulnerabilities are among the most experienced developers in the project. More precisely, in Apache HTTP there are just 44 developers who patched software vulnerabilities, while in Apache Tomcat there are only 10. Moreover, there is a negligible difference in terms of experience between the developers who fix vulnerabilities and the developers with the maximum experience at that point in the history of the project. Nevertheless, there are still cases in which vulnerabilities are fixed by developers less experienced than the average (20% for Apache HTTP and 4% of the cases for Apache Tomcat).

We report in Figure 3 the distributions of the experience of the developers who fix vulnerabilities and the one of the average team member. In both cases, the first quartile of the vulnerability fixers is close to the third quartile of the average team members. It is worth remarking again that the experience of vulnerability fixers in Apache Tomcat is generally higher. We discuss this more in depth in Section IV.

Figure 4 shows in what percentage both the projects are modified by (i) only owners, (ii) only non-owners, (iii) both. It is clear that the developers who fix vulnerabilities are usually not owners of the files they modify in most of the cases for Apache HTTP (80%). Also in Apache Tomcat the majority

[3] https://dibt.unimol.it/report/cve-icst2019/

TABLE IV: Experience (number of commits) of vulnerability fixers.

Project	Vulnerability Fixers	Others	Difference	Significance (p-value)	Effect Size
Apache HTTP Server	645.4	197.8	447.6	< 0.001	0.55 (large)
Apache Tomcat	5,294.5	374.0	4,920.5	< 0.001	0.70 (large)

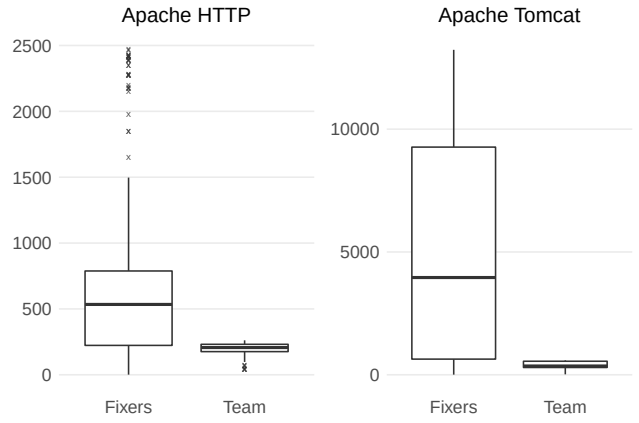

Fig. 3: Vulnerability fixers' *vs.* mean team's experience.

of CVE Entries are fixed by non-owners, even if this happens less frequently (41%). This means that there are developers specialized in fixing vulnerabilities that change the files even if they are not the most experienced developers for those files.

We analyzed more in depth this phenomenon. We considered the files appearing in each CVE Entry and we computed, for each of them, how many times it was modified by the author of the patch at that point in time. We report in Table V the distribution of the percentage of changes that the developers who fix vulnerabilities did to the file before the patch. We confirm that most of them rarely contributed to the files. For Apache HTTP, we found that in 531 file changed during the fixes of vulnerability out of 855, the fixers accounted for less than 10% of the total changes. For Apache Tomcat, instead, this scenario, *i.e.*,, the fixers that accounted for less than 10% on that files of the total changes, was observed on 237 files of 877.

Combining both the results, we can conclude that vulnerability fixers are experienced developers who generally do not own the files they modify. This result is counter-intuitive: since we measure both experience and ownership in terms of number of changes done by the developers, it would be reasonable to think that experienced developers are more likely to own many files of the project. Therefore, the probability of having fixes involving files owned by the vulnerability fixers should be high. This is not the case for Apache HTTP: only in less than 20% of the cases the vulnerability fixers modify at least a file they own. However, on Apache Tomcat 41% of the fixers have been performed by non-owners of the modified files.

We also analyzed the patches in the mailing lists: we found 40 for Apache HTTP and only 5 for Apache Tomcat. This is in line with what we observed, *i.e.*, Apache HTTP uses a more traditional open-source approach to fix vulnerabilities. The authors of such patches are 26 for Apache HTTP and 4 for Apache Tomcat. We found that 48% of the authors of patches for Apache HTTP became committers of the project later; for Apache Tomcat, instead, only one of the four authors became an actual contributor.

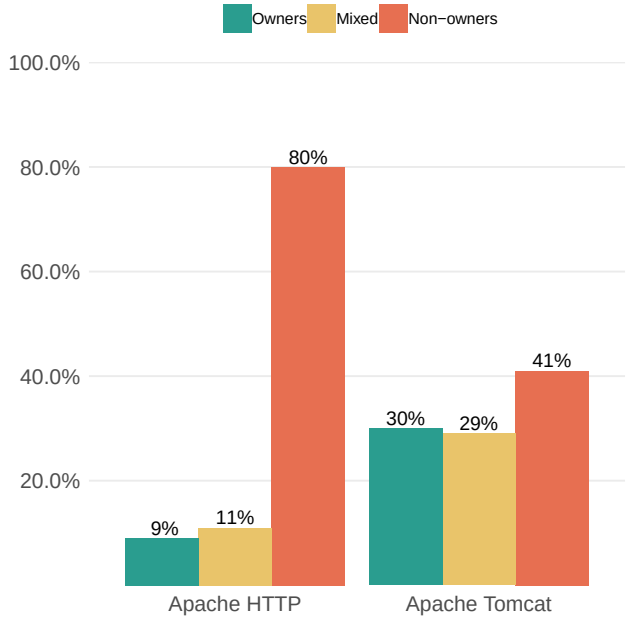

Fig. 4: Vulnerability fixers' ownership of modified files.

TABLE V: Distribution of the relative number of changes by patchers to the files involved in CVE Entries patches.

Project	Min.	Q1	Median	Q3	Max.
Apache HTTP Server	< 0.1%	1.2%	4.5%	25.0%	100.0%
Apache Tomcat	< 0.1%	9.2%	36.3%	100.0%	100.0%

Summary of RQ$_1$. The developers who fix software vulnerabilities are very experienced and they usually are not the owners of the files affected by the vulnerabilities.

B. *RQ$_2$: When are vulnerabilities fixed?*

Figure 5 shows the distribution of the days lapsed between the publication date of the CVE Entry and the date of the last commit for both the projects.

First, it is worth noting that the majority of the values are negative. This means that the CVE Entry was disclosed *after* the vulnerability was fixed. This is the desirable scenario, because an attacker would not be able to use the information

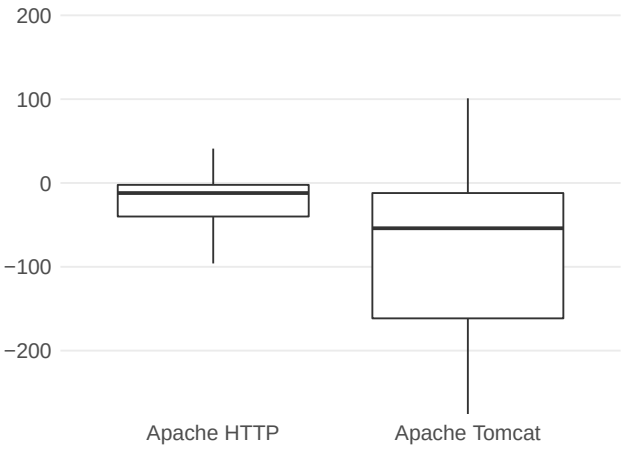

Fig. 5: Days of exposure of vulnerabilities (without outliers).

given in the CVE Entry. The median of both the distributions is negative: -12 days for Apache HTTP and -54 days for Apache Tomcat. This means that CVE Entries about the vulnerabilities that affect Apache HTTP become CVE Entries in less than two weeks, while the ones affecting Apache Tomcat need almost two months. This could mean that (i) CVE Entries are published faster for Apache HTTP, or that (ii) vulnerabilities are fixed faster in Apache Tomcat.

There is a non-negligible number of cases in which the vulnerabilities were fixed only after the related CVE Entry was published: for Apache HTTP there are 28 of such cases (20.2%), while for Apache Tomcat there are only 7 of them (6.3%). The most interesting aspect, though, is that some vulnerabilities require a very long time to be fixed. We manually analyzed some of the cases in which the days of exposure of the vulnerabilities was higher than 10 days.

For Apache HTTP, the CVE Entry with the longest exposure time was **CVE-2003-1418**. Such a vulnerability was never officially fixed in Apache HTTP: a RedHat developer declared that it "*poses no threat to the target machine running httpd*" [4]. However, we found a commit [2] that partially addresses it after about 7 years since the publication date. It is worth noting that the CVE Entry reports that such a vulnerability has a *partial* confidentiality impact "*There is considerable informational disclosure*" and the assigned CVSS score is 4.3.

Another interesting case is the vulnerability reported in **CVE-2003-0132**. Most of the fixing commits were done before the publishing date of the CVE Entry. However, two commits were done after the publishing date: in one of them, there was just reformatting of the code (indentation adjustment), but in the other one [1] it was addressed a corner case which could actually result in the vulnerability exploit. The fix of other vulnerabilities, instead, was simply delayed for no specific reason. The related CVE Entries are:

- **CVE-2001-0925** (fixed after 148 days);
- **CVE-2003-0020** (first commit after 251 days, completely fixed after 300 days);
- **CVE-2007-6420** (fixed after 147 days);
- **CVE-2013-5704** (fixed after 91 days).

As for Apache Tomcat, we found that, in some cases, the longer exposure time was due to the fact that the fix did not cover all the corner cases (CVE-2008-2938, CVE-2007-0450 and CVE-2007-3385, with 4 years of exposure for the first two and 101 days for the third one); however, according to the authors of the patches, such corner cases are not exploitable. In another case (CVE-2008-3271, with 50 days of exposure) we could not find any specific reason for the delay.

We also found few vulnerabilities with *incomplete* patches. For Apache HTTP we found 5 of such vulnerabilities (*i.e.*, 3.5%), while for Apache Tomcat only 3 (2.6%). In most of the cases, the CVE Entries related to such vulnerabilities were published only after the actual vulnerability was fixed, which slightly reduces the risks of exploit. Nevertheless, these are still very dangerous cases: a release is done with vulnerable code apparently fixed; such a fix is reported in the change log of the project; the released version could be installed in some production environments; in the meanwhile, an attacker could know about the vulnerability, check if the patch was complete and, if not, try to exploit it whatsoever. It is worth noting that updating software like Apache HTTP and Tomcat in a production environment requires the restart of the service, causing a downtime of the web applications. Therefore, the update of such components is reasonably done not too often, in order to avoid such a problem [17]. This further increases the risk connected to incomplete patches and the cost of updating such releases in production.

Finally, Figure 6 shows the distribution of the days lasted since the first to the last commit of the fix. Typically, such a time is short (8 days median for Apache HTTP and 5 for Apache Tomcat). However, also in this case there are several outliers, and, in the worst cases, the number of days lasted is very high. We observed that the biggest delay was for **CVE-2008-2938**, reporting a vulnerability that affected Apache Tomcat. The core of such a vulnerability was fixed in 2008, but two additional commits were done in 2013 (after 4 and a half year) to cover some corner cases [3]. Such commits introduce a test case that finds a failure in the program. However, the author of the commit highlights in the commit message that such a failure is not exploitable. Indeed, the original patch is officially considered as *complete*, since the vulnerability only affects a single version of the software.

Summary of RQ$_2$. Most of the vulnerabilities are fixed before their CVE Entry is published; however, we observed some potential threats due to incomplete fixes.

C. RQ$_3$: How are vulnerabilities fixed?

Table VI shows some statistics about the number of commits, files and lines modified for each vulnerability fix.

It can be seen that the typical vulnerability fix is done with few commits (less than 2, usually). Also, the number of files modified is very small: a typical patch for a vulnerability

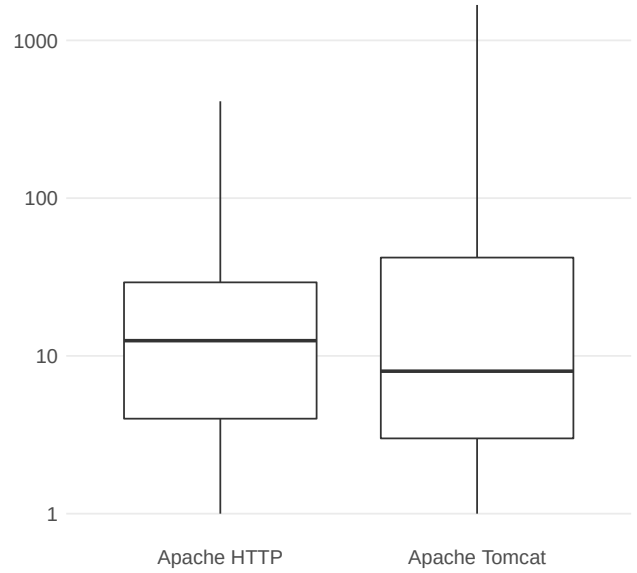

Fig. 6: Days lasted since the first to the last commit when more than a commit was done (without outliers).

TABLE VI: Statistics on the number of commits done and files/lines changed in a vulnerability fix.

Commits					
Project	**Min.**	**Q1**	**Median**	**Q3**	**Max.**
Apache HTTP Server	1	1	1	2	60
Apache Tomcat	1	1	1	2	21
Files					
Project	**Min.**	**Q1**	**Median**	**Q3**	**Max.**
Apache HTTP Server	1	2	2	5	65
Apache Tomcat	1	2	4	7	162
Lines					
Project	**Min.**	**Q1**	**Median**	**Q3**	**Max.**
Apache HTTP Server	1	12	45	112	3,065
Apache Tomcat	1	19	71	218	9,846

involves few files (median 2 and 4) and also few lines (median 45 and 71). However, some vulnerabilities required many changes and they involved many files and lines of code.

We report in Figure 7 the distribution of the file types involved in the vulnerability fixes. As expected, most of the changes involve source code (70% for Apache HTTP and 66% for Apache Tomcat). We observed that developers often report the fix in the change logs of the project (*i.e.*, `CHANGES` for Apache HTTP and `changelog.xml` for Apache Tomcat).

We also checked how many test files (*i.e.*, test cases or test data, such as JSP pages for Tomcat) were added or modified in the process of vulnerability patching. For Apache HTTP we found 2 test files, each of them appearing in the patch of a single CVE Entry. Instead, for Apache Tomcat we found (i) 12 test files appearing in the patch of a single CVE Entry, (ii) 3 test files appearing in the patch of two CVE Entries, and (iii) a test file appearing in the patch of three CVE Entries.

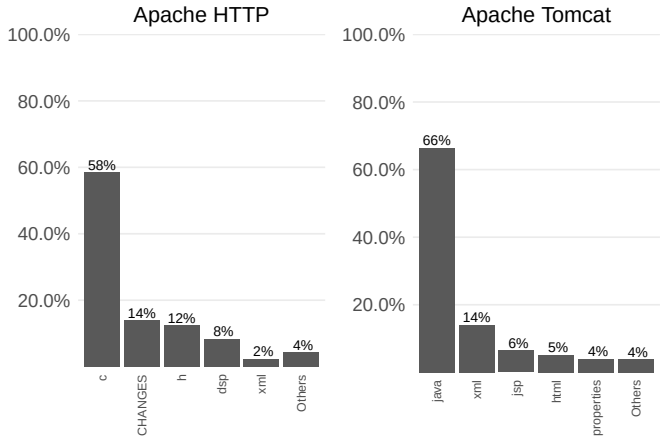

Fig. 7: Distribution of the extensions of the file changed in vulnerability fixes.

TABLE VII: Locality of the changes.

Project	**Local changes**	**Global changes**	**No code involved**
Apache HTTP	68.8%	28.3%	2.9%
Apache Tomcat	46.8%	44.1%	9.0%

Table VII shows the locality of the changes made to fix the vulnerabilities in the two projects. In both cases, local changes are more frequent. This is more evident for Apache HTTP (68.8%), while for Apache Tomcat they are almost equal. This can be due to the fact that Java packages are usually smaller than C modules. An interesting fact is that a quite large number of vulnerabilities are fixed without changing any line of source code. We analyzed such cases:

- for Apache HTTP, the fix involved build configuration (CVE-2003-0017), writing better documentation to avoid configuration errors (*i.e.*, CVE-2012-0883 and CVE-2006-4110);
- for Apache Tomcat, instead, such vulnerabilities regarded insecure example applications (*e.g.*, CVE-2006-7196 and CVE-2007-1355), issues with default configuration (*e.g.*, CVE-2007-5342 and CVE-2009-3548), vulnerabilities in dependencies (CVE-2011-2729) or other scripts (CVE-2017-15706).

Summary of RQ$_3$. Vulnerability patches are usually small, they require few commits and they are local. Surprisingly, some fixes do not even require the modification of source code.

IV. Discussion and Lessons Learned

The results presented in Section III allow us to make interesting conclusions about how vulnerability fixing is performed.

First, there are patterns that repeat in both the projects: experienced developers who are not always very experienced with the files with the vulnerabilities are the ones that, more likely,

fix the vulnerabilities; CVE Entries are generally published after the vulnerability is fixed; vulnerabilities are fixed with few commits, and they generally involve a low number of files/lines.

We can conclude that it is *easy* to create such patches (because they are short), but it is *hard* to spot them (because experienced developers are needed). Also, we observed that vulnerabilities are hard to test: few of them require additional commits that are done even years after the original patch was released. While this may be true also for normal bugs, in this context it should be critical to have complete patches in a short time and, above all, before the information is public.

Lesson 1. Software vulnerabilities are quite easy to fix, but hard to spot and to test.

While for both the projects the vulnerability fixers are more experienced than the average, we found that in Apache Tomcat such a difference is more extreme. Besides, we also observe that both the exposure and fix time in such a project is generally lower. There are many factors that may affect this, such as the programming language. However, we observed that Apache Tomcat has a very small and active security team that handles the vulnerabilities (10 developers) compared to Apache HTTP (44 developers).

We also found that in Apache Tomcat most of the vulnerabilities were fixed by a single developer, who is a security expert. The high level of expertise of vulnerability fixers both on the project and on security aspects might be the reason why Apache Tomcat seems more reactive in the fixing process.

Lesson 2. Having a security team with few and experienced developers seem to benefit open source projects' vulnerability fixing process.

The distribution of the files modified depicted in Figure 7 shows that developers usually document the vulnerabilities they fix in the change logs. However, this does not always happen. Generally, we found quite difficult tracking some vulnerabilities to the commits done to patch it. This is true above all for older CVE Entries, but we found also new CVE Entries affected by such a problem. Having explicit links between CVE Entries and fixing commits may help both researcher, who would be able to devise more easily approaches to automatically fix vulnerabilities, and practitioners, who would be able to easily browse similar vulnerabilities to find past solutions to similar present problems.

Lesson 3. Information in the vulnerability databases are often lacking a link to the fixing commits.

V. Threats to Validity

Threats to *construct validity* regard the relationship between theory and observations. The main threat of such a kind in our study is due to the dataset building, *i.e.*, to the linking between CVE Entries and related fixing commits/patches. To minimize the risk of having commits not actually related to a CVE Entry or to miss commits, we relied on manual analysis to perform such a task.

Threats to *internal validity* concern internal factors of our study that could influence the results. To answer RQ_1 we rely on "number of commits" to measure the experience of developers and on "number of changes" to decide the ownership of a file. It could happen that some developers prefer to make commits more frequently than others (regardless the number of changes actually made in the project): this would result in a larger experience in our dataset, not actually present in reality. As for the ownership, it can happen that many developers made similar number of changes to a file, but we assume that only a developer is the owner of each file. However, in general the number of commits represents a good proxy of developer's experience [18].

Moreover, to answer RQ_2 we measure the exposure time, the time needed for the fix and we distinguish *complete* from *incomplete* fixes. Both exposure and fix time are *lower bounds*, *i.e.*, the actual exposure and fix time is larger or equal to the one we report. Indeed, for the exposure time we assume that the only way an attacker can know about a vulnerability is through CVE Entries, which is not always the case. To mitigate this threat, we report also the number of vulnerabilities not completely fixed. This allows us to keep into account another potential scenario in which vulnerabilities are disclosed before the actual fix is released. As for the fix time, we ignore the time needed to make the first commit. We did this because there is no reliable way of estimating such a time.

Threats to *external validity* concern the generalizability of our results. We considered only two projects from the Apache ecosystem. It is worth remarking that the manual analysis needed to build our dataset makes such a study hardly scalable. Furthemore, it is worth remarking that both the systems we took into account are two projects from the Apache ecosystem. Since such projects may have similar governance rules and contribution guidelines the dataset may not be very representative and may influence our results. We choose, however, to study the two projects with the highest number of vulnerabilities in the Apache ecosystem. We expect differences in other projects. We plan to cover more projects from the Apache ecosystems in future work.

VI. Related Work

The most related work is a large-scale empirical investigation done by Li and Paxson [10]. In their study, the authors built a large dataset containing more than 3,000 CVE Entries and the related fixing commits using an automated approach. The authors report an analysis on various aspects of vulnerability fixing from the security perspective. Such analysis include (i) the duration of the impact of vulnerabilities, (ii) the reliability of the fix, (iii) the difference with non-security fixes. Our investigation is more from an engineering perspective. We focus our analysis on the developers who produce the patches, on the patching timing and process. We could not use the dataset by Li and Paxson because it is not publicly available. Even if our dataset is much smaller, it was manually built by

analyzing the commits possibly involved in the fix to achieve a high level of completeness, *i.e.*, to minimize the likelihood of missing patching commits for the CVE Entries or to include non-related commits.

Shahzad *et al.* [19] report a large-scale explorative study on the vulnerability life-cycle. They report insightful trends, such as the fact that vulnerabilities exploitable from remote grew to more than 80% of the total in 2011. They also show that patching in closed-source software is usually faster compared to open-source software. In our work we studied more in depth the patching practices focusing only on open-source projects, and our results suggest that a vulnerability patching process more similar to the one used in closed-source software (*i.e.*, the one used in Apache Tomcat) is more effective than a typical open-source development approach (*i.e.*, the one used in Apache HTTP).

Huang *et al.* [9] analyzed 131 patches from five open-source projects, and they showed that, in some cases, developing a patch required much time and it was error-prone. They use such results to justify the introduction of a new approach for minimizing the patch delay. Moreover, Nappa *et al.* [17] studied the patch deployment process in ten client applications. They found that only 28% of the patches reach 95% of the hosts in the observation period. Differently from them, we focus on the engineering process of vulnerability fixing.

Other studies focused on the relationship between developers characteristics and the introduction of vulnerabilities. Meneely *et al.* [11] investigated the influence of Linus' Law on security, analyzing code review participation. Linus' Law is declared by Eric Raymond as "many eyes make all bugs shallow". More specifically, this study analyzed the association between collaborative reviews and vulnerabilities that were missed by the review process. Their results show key risk factors of using Linus' Law with vulnerabilities, namely the lack of security experience and lack of collaborator familiarity. In our work, we focus on vulnerability patching instead of vulnerability introduction. However, similarly to Meneely *et al.* [11], we also consider the authors of the patches in our analysis.

Camilo *et al.* [6] performed an in-depth analysis of the Chromium project to examine the relationship between bugs and vulnerabilities. They collected bugs and post-release vulnerabilities over five Chromium releases on six years of development. They examined how various categories of pre-release bugs and review experiences are associated with post-release vulnerabilities. Their results indicate that bugs and vulnerabilities are empirically dissimilar groups. Our work differs from this study because we focused on vulnerability fixing instead of vulnerability introduction.

Several studies have found a consistent statistical association between metrics that quantify the developer activity and the introduction of vulnerabilities [12]–[16]. These studies regarded the development of socio-technical metrics with various techniques, *i.e.*, interactive churn [16], developer networks [14], [15], and developer network clustering [13].

Bosu [5] analyzed how the experience affects the effectiveness of the code reviews in terms of vulnerabilities. The author used 10 popular Open Source projects to identify vulnerable code changes. The results show that the contributions of inexperienced developers are more likely to lead to the introduction of software vulnerabilities.

Finally, Yin *et al.* [21] studied and characterized incorrect bug-fixes. They analyzed both *error patterns* and *human reasons* behind the development of incorrect patches. In this study we also analyze incorrect patches, but we focus on software vulnerabilities only.

VII. Conclusion and Future Work

We presented an empirical investigation on the vulnerability fixing process by analyzing the complete history of two open-source projects, Apache HTTP and Apache Tomcat. We studied such a phenomenon from three different perspectives: *who* fixes software vulnerabilities, *when* it happens, *how* it is done. We built a dataset containing 239 software vulnerabilities, manually linking them to the fixing commits in the revision history of the projects, for a total 610 commits.

Our results show that developers who work on software vulnerabilities are much more experienced than the average. Moreover, they usually do not modify the files they own. We also showed that vulnerabilities are mostly fixed before they are publicly disclosed. However, we found and analyzed some examples in which vulnerabilities were not completely fixed after the vulnerability was disclosed. Some vulnerabilities still received fixing commits 4 years after the CVE Entry was published and the partial patch was implemented and released.

Finally, we observed that most of the vulnerability patches are trivial, *i.e.*, few files and lines modified. Also, the developers usually make few commits to fix vulnerabilities. However, also in this case we found some exceptions.

The results of our study open interesting and novel research directions. For instance, we found that vulnerability patches were not complete in about 3% of the cases. Also, we showed that testing security patches is not always easy. Thus, an approach able to automatically understand when a patch is *partial*, either by generating test cases or by using machine learning, is particularly useful to mitigate the high danger of partial patches.

We also observed that developers document their vulnerability fixes, but they are not supported in doing that. Most likely, an automated approach that recognizes and documents such changes would be welcome.

Finally, we found that vulnerability databases often lack explicit links to the fixing commits. In our study we manually defined such links. An automated approach with this purpose would greatly benefit both researchers and practitioners.

Addressing some or all the above issues is part of our agenda for future work. In addition, our data suggest that a small and much more experienced security team can benefit the whole vulnerability fixing process. We plan to verify this trend with a larger empirical study involving many different software projects.

REFERENCES

[1] Apache http fix of cve-2003-0132. https://git.io/fhV6S. Accessed: 2018-10-11.

[2] Apache http fix of cve-2003-1418. https://git.io/fhV6y. Accessed: 2018-10-11.

[3] Apache tomcat fix of cve-2008-2938. https://git.io/fhV69. Accessed: 2018-10-11.

[4] Redhat cve-2003-1418. https://access.redhat.com/security/cve/cve-2003-1418. Accessed: 2018-10-11.

[5] A. Bosu. Characteristics of the vulnerable code changes identified through peer code review. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 736–738. ACM, 2014.

[6] F. Camilo, A. Meneely, and M. Nagappan. Do bugs foreshadow vulnerabilities?: a study of the chromium project. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, pages 269–279. IEEE Press, 2015.

[7] N. Cliff. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological Bulletin*, 114(3):494, 1993.

[8] R. Grissom and J. Kim. *Effect Sizes for Research: A Broad Practical Approach*. Lawrence Erlbaum Associates, 2005.

[9] Z. Huang, M. DAngelo, D. Miyani, and D. Lie. Talos: Neutralizing vulnerabilities with security workarounds for rapid response. In *2016 IEEE Symposium on Security and Privacy (S&P)*, pages 618–635. IEEE, 2016.

[10] F. Li and V. Paxson. A large-scale empirical study of security patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2201–2215. ACM, 2017.

[11] A. Meneely, A. C. R. Tejeda, B. Spates, S. Trudeau, D. Neuberger, K. Whitlock, C. Ketant, and K. Davis. An empirical investigation of socio-technical code review metrics and security vulnerabilities. In *Proceedings of the 6th International Workshop on Social Software Engineering*, pages 37–44. ACM, 2014.

[12] A. Meneely and L. Williams. Secure open source collaboration: an empirical study of linus' law. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 453–462. ACM, 2009.

[13] A. Meneely and L. Williams. Strengthening the empirical analysis of the relationship between linus' law and software security. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, page 9. ACM, 2010.

[14] A. Meneely and L. Williams. Socio-technical developer networks: Should we trust our measurements? In *Proceedings of the 33rd International Conference on Software Engineering*, pages 281–290. ACM, 2011.

[15] A. Meneely, L. Williams, W. Snipes, and J. Osborne. Predicting failures with developer networks and social network analysis. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 13–23. ACM, 2008.

[16] A. Meneely and O. Williams. Interactive churn: Socio-technical variants on code churn metrics. In *Int'l Workshop on Software Quality*, pages 1–10, 2012.

[17] A. Nappa, R. Johnson, L. Bilge, J. Caballero, and T. Dumitras. The attack of the clones: A study of the impact of shared code on vulnerability patching. In *2015 IEEE Symposium on Security and Privacy (S&P)*, pages 692–708. IEEE, 2015.

[18] F. Rahman and P. T. Devanbu. Ownership, experience and defects: a fine-grained study of authorship. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 491–500. ACM, 2011.

[19] M. Shahzad, M. Z. Shafiq, and A. X. Liu. A large scale exploratory analysis of software vulnerability life cycles. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 771–781. IEEE, 2012.

[20] R. Telang and S. Wattal. An empirical analysis of the impact of software vulnerability announcements on firm stock price. *IEEE Transactions on Software Engineering*, (8):544–557, 2007.

[21] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram. How do fixes become bugs? In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 26–36, New York, NY, USA, 2011. ACM.