Noname manuscript No.
(will be inserted by the editor)

The Secret Life of Software Vulnerabilities: An Investigation Into the Lifecycle of Vulnerabilities

Larissa Braz · Roberta Guadagni · Fabio Palomba · Filomena Ferrucci · Alberto Bacchelli

Received: date / Accepted: date

Abstract Software vulnerabilities are potential weaknesses in source code that may be exploited to cause loss or harm. While researchers have been devising a number of methods to deal with vulnerabilities, there is still a noticeable lack of knowledge on their lifecycle, for example how vulnerabilities are introduced and removed by developers. This information can be exploited to design more effective methods for vulnerability prevention and detection as well as to understand the granularity that these methods should aim at. To investigate how, when, and under which circumstances vulnerabilities *are introduced* in software projects, as well as whether, after how long, and how they *are removed*, we analyze 1,195 vulnerabilities with public patches from the National Vulnerability Database, pertaining to 616 open source software projects on GitHub, including OpenSSL and Linux. In particular, we define a six-step process that involves both automated parts (for example, using the SZZ algorithm to find the vulnerability-inducing commits) and manual analyses (how the patches were removed). The investigated vulnerabilities can be classified in 62 categories, take on average 8,248 commits in a file before being introduced, and remain unfixed for a median of 1,546 commits and 1,043 days. Most vulnerabilities are introduced while developing new features and removed through small changes. Our results have implications on when and how vulnerability detectors should work to better assist developers in early detecting these issues.

Larissa Braz, Fabio Palomba, Alberto Bacchelli
University of Zurich
E-mail: larissa@ifi.uzh.ch, palomba@ifi.uzh.ch, bacchelli@ifi.uzh.ch

Roberta Guadagni, Filomena Ferrucci
University of Salerno, Italy
E-mail: r.guadagni1@studenti.unisa.it, fferrucci@unisa.it

1 Introduction

Software vulnerabilities are potential problems in the design, implementation, or operation management of a software system that can be used to break through security policies [78], possibly causing loss or harm [38, 41, 72]. The risks associated with vulnerabilities on software systems are extreme: For instance, in 2017, the WannaCry Ransomware Attack [18] exploited a vulnerability to infect more than 200,000 computers across over 150 countries within a day, with total damages in the range of hundreds of millions USD. Perhaps more importantly, it is estimated that by 2021 vulnerabilities will cost to businesses and users over 6 trillion USD [10] and will affect an even larger variety of software systems, ranging from public physical tests [20] to data-intensive applications [29], and more. For these reasons, keeping software security under control is still one of the main concerns of developers and organizations [21,60].

Nowadays, to address this concern software development organizations and teams are adopting the McGraw's [56] advice to *"build security in"* rather than waiting until vulnerabilities are discovered in running software [62]. At the same time, researchers have been proposing novel methods and tools to assist developers and facilitate the identification of software vulnerabilities [22, 49, 54, 55]. Nevertheless, a number of recent empirical studies [33, 35, 53, 63, 66] provided evidence that, despite developers are concerned with vulnerabilities and their potential effects, these still often occur in practice.

A possible reason behind the diffuseness of vulnerabilities may be the lack of empirical knowledge of *how* they actually manifest themselves, i.e., when developers introduce them, during which development activities vulnerabilities are more prone to be introduced, or how developers remove them in practice. An improved understanding of these aspects may inform software engineering community and tool vendors on how to better support developers in both the vulnerability identification and fixing process [83], for instance by devising novel mechanisms or adapting currently available detection approaches so that they can be ran in the moment of the development when vulnerabilities are usually introduced (e.g., at commit-level). Moreover, the exploratory analysis of vulnerability life-cycle can uncover interesting patterns that would be helpful for the deployment of best practices [77] that help improving the development process for security [61].

To bridge such a knowledge gap and provide researchers and practitioners with a comprehensive overview of how software vulnerabilities are introduced and removed, in this paper we present a large-scale empirical study on the life-cycle of software vulnerabilities. To conduct our investigation, we first mine the National Vulnerability Database (NVD) [11] (the U.S. government repository of standards-based vulnerability management data) in order to extract a set of vulnerabilities for which their fixing commits are public. Overall, we study 1,195 vulnerabilities of 62 categories across 616 different GitHub projects. Then, we automatically identify the corresponding vulnerability-inducing commits (verifying the performance of the automated solution) and investigate (i) when vulnerabilities are introduced, (ii) by whom, (iii) under which circum-

stances, (iv) how long they remain in the considered software systems, and (v) how the fixing process takes place.

The key results of our empirical study show that developers mostly introduce vulnerabilities while developing new features. Furthermore, they remain in a project's codebase for on median 1,546 commits and 1,043 days, and developers remove them through simple changes in code, such as escaping functions of HTML entities. Finally, we find that expert developers are those that are more prone to introduce vulnerabilities.

Our findings provide evidence that developers, even the most expert ones, do need help to detect vulnerabilities earlier, for example, through a better tool support. In addition, vulnerability detectors should be context-dependent and take into account what a developer is doing when suggesting potential vulnerable code. To sum up, the main contributions of this paper are:

1. Empirical evidence on when and under which circumstances vulnerabilities are introduced, their survivability, and their removal, based on a large-scale empirical study on the complete life-cycle of 1,195 real vulnerabilities pertaining to 616 different projects;
2. A large curated dataset [13] on software vulnerabilities with details on when and under which circumstances they were introduced, and their survival time. This publicly available source can be exploited by the software engineering community to build upon our research and further explore the problem of software vulnerabilities.

Structure of the paper. We organize the remainder of this article as follows. In Section 2, we describe the design of our empirical study. In Sections 3 and 4, we present and discuss the results in details and the lessons learned, respectively. Finally, we present the related work in Section 5, and the concluding remarks in Section 6.

2 Methodology

This section describes the research goals driving our empirical study, our terminology, the data extraction and validation procedure, as well as the research method to answer each research question. We conclude presenting the threats to the results' validity.

2.1 Research Goals

The *goal* of the study is to investigate the life-cycle of software vulnerabilities, with the *purpose* of assessing when they are introduced and fixed by developers as well as the circumstances and reasons behind vulnerabilities introductions. The *perspective* is of both researchers and practitioners: the former are interested in better understanding the phenomenon of software vulnerabilities,

from their introduction to removal, with the aim of devising novel identification and refactoring techniques that can provide an improved support to developers; the latter are interested in gathering information on the circumstances that most likely introduce vulnerabilities, so that they could inform their development process.

Our study is structured around four main research questions (**RQ**s). We start analyzing *when* vulnerabilities are introduced in source code, particularly to understand whether the introduction takes place with the creation of a new file or during maintenance/evolution activities performed on existing files. Such an analysis may reveal insights on the *moment* in which vulnerability detectors should give feedback to developers [86]; for example, should the vulnerabilities be mainly introduced when the affected file is created, a *just-in-time* approach pinpointing potential vulnerabilities at commit-time would be worthwhile [50, 70]. This leads to our first research question:

> **RQ$_1$.** *When are vulnerabilities introduced: during a new file creation or during maintenance/evolution activities on existing files?*

In the second place, we investigate under which circumstances vulnerabilities are more likely to be introduced by developers. We focus on three aspects: *commit goal*, i.e., the action that a developer usually does when introducing vulnerabilities (e.g., new feature implementation or refactoring activity); *project status*, i.e., the proximity of a vulnerability introduction commit to a new release or the startup of the project; and *developer status*, i.e., whether a programmer introducing a vulnerability has usually a high workload or whether s/he is a newcomer. These three aspects are key to describe *context* and *situations* in which developers are more prone to introduce vulnerabilities, two pieces of information that may be exploited by vulnerability detectors and refactoring recommenders to output more precise suggestions [51, 75], but also to provide developers with contextual details that can help understanding the reasons leading to the vulnerability introduction [52, 58]. Hence, we ask:

> **RQ$_2$.** *Under which circumstances are vulnerabilities introduced?*

After describing the mechanisms leading to the introduction of vulnerabilities, we move toward the understanding of their *longevity* by means of a survival analysis method [59], namely we investigate how long they remain in source code. This analysis helps describing the life of vulnerabilities: on the one hand, this is useful to understand for how long such vulnerabilities can be exploited by externals, thus challenging previous findings in the field [77]; on the other hand, we can gather insights on the *reaction time* of developers, which may suggest the need for additional instruments to alert practitioners of the presence or the potential impact of unfixed vulnerabilities [24, 28]. Thus, we ask our third research question:

> **RQ$_3$.** *What is the survivability of vulnerabilities?*

Finally, we investigate how vulnerabilities are removed, (e.g., do developers completely remove the vulnerable code or apply specific refactoring operations depending on the type of the vulnerability encountered?). Such an analysis can help researchers in understanding how to better support developers with (semi-)automated techniques to remove vulnerabilities or to mitigate their effect [25, 37, 39]. Therefore, we ask:

RQ$_4$. *How are vulnerabilities removed?*

2.2 Context Selection

The *context* of the study is composed of vulnerabilities and change history information of the affected software projects.

Vulnerabilities: In this work, we analyze vulnerabilities from the National Vulnerability Database (NVD) [11]. This is a repository that was originally created in 2000 by the U.S. NIST Computer Security Division [17] to collect and provide public information about known vulnerabilities affecting software systems and their causes. We rely on this database for three main reasons: (1) It includes a comprehensive set of information on vulnerabilities, such as references to security checklists, security-related software flaws, misconfigurations, product names, and additional information such as impact metrics (Common Vulnerability Scoring System - CVSS), vulnerability types (Common Weak Enumeration - CWE), applicability statements (Common Platform Enumeration - CPE); (2) It is continuously improved, because the NVD updates the database entries (called "Common Vulnerabilities and Exposure" - CVEs) that have been changed [11]; and (3) It is a standard de-facto for the research community [19, 48, 57, 90].

NVD provides a set of 2,973 vulnerabilities and their data. Among this set, we select those for which both the fixing commit and the project's GitHub are available (see next point). As a result, we take into account 1,195 vulnerabilities of 62 distinct types. We use their fixing commits to identify the vulnerability inducing commits, which allows us to analyze how vulnerabilities are introduced, by whom, and under which circumstances. We also compute information on the vulnerability survivability. Table 1 presents the types of the top-15 most common analyzed vulnerabilities; the complete list is available in our online appendix [13]. Interestingly, a total of 82 of the studied vulnerabilities are not classified by kind (i.e., they are assigned to the *NVD-CWE-Other* category): this may indicate that further studies into vulnerability types and characteristics may be worthwhile [30].

Systems Histories: We gather the code history of the projects in which each subject vulnerability appeared. In total, we study vulnerabilities pertaining to 616 different projects. All the analyzed systems are hosted on GitHub [5].

Table 1: Top-15 most frequent vulnerabilities in our dataset, by CWE type.

CWE	Kind	Total
CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	213
CWE-79	Improper Neutralization of Input During Web Page Generation	209
CWE-20	Improper Input Validation	99
NVD-CWE-Other	Other	69
CWE-200	Information Exposure	65
CWE-264	Permission/Privileges/Access Controls Issues	63
CWE-125	Out-of-bounds Read	52
CWE-89	Improper Neutralization of Special Elements used in an SQL Command	51
CWE-476	NULL Pointer Dereference	46
CWE-399	Resource Management Errors	38
CWE-352	Cross-Site Request Forgery	37
CWE-284	Improper Access Control	31
CWE-22	Improper Limitation of a Pathname to a Restricted Directory	24
CWE-190	Integer Overflow or Wraparound	22
CWE-189	Numeric Errors	20
CWE-416	Use After Free	19
CWE-787	Out-of-bounds Write	19
CWE-94	Improper Control of Generation of Code	18
CWE-310	Cryptographic Issues	18
CWE-254	7PK - Security Features	17

GitHub is a hosting service for Git repositories, it allows us to access the Git history of the projects and study the life-cycle of the vulnerabilities. Table 2 provides information on the top-15 projects by number of vulnerabilities. Our appendix reports the complete list of the projects [13].

2.3 Terminology

Next, we define the terms used in the following sections to explain our data extraction process and answer our research questions:

– A vulnerability v_i belongs to the NVD set of vulnerabilities V;

Table 2: Top-15 projects with more vulnerabilities in our dataset. Commits = Number of commits in GitHub; Files = Num. of files after the latest commit; KLOC = LOC after the latest commit; Developers = Number of distinct committers; Vulnerabilities = Number of analyzed vulnerabilities.

Project	Commits	Files	KLOC	Developers	Vulnerabilities
TCPdump	5,577	264	100	150	60
PHPmyadmin	115,032	1,565	1,102	1,600	49
FFmpeg	93,463	3,903	1,178	1,750	48
ImageMagick	15,441	1,017	535	71	44
WordPress	39,252	1,832	622	90	30
Radare2	21,086	2,319	665	766	30
KRB5	19,542	1,828	381	123	25
PHP-src	111,636	2,270	1,215	962	24
Jenkins	28,060	3,007	253	811	21
Kanboard	3,854	2,496	191	319	19
File	3,381	68	17	9	19
Exponent-cms	8,602	5,902	1,099	15	16
Dolibarr	74,858	6,355	1,452	380	14
phpMyFAQ	9,132	390	82	61	13
LibTiff	3,006	428	140	16	13

- Each v_i occurs in a project hosted in a repository r_i;
- Each r_i belongs to the set of Git repositories R hosted in GitHub;
- Each r_i has a set of commits C_{r_i};
- Each v_i has at least one vulnerability inducing commit $ic_i \in C_{r_i}$;
- Each v_i has a fixing patch $p_i \in C_{r_i}$;
- Each ic_i and each p_i has at least a touched file f_i; and,
- Each f_i belongs to the set of touched files F.

2.4 Vulnerability Life-cycle Example

In this section, we show a short example of a vulnerability's life-cycle to demonstrate from which commits and files we collect data during our process (see Section 2.5). When filtering vulnerabilities from NVD, we identify vulnerabilities with fixing patches (Step 1) and with existing GitHub repositories (Step 2). Figure 1 presents an example of a project's repository, where each stroke in the life-line represents a different commit. In this example, the last commit removed the analyzed vulnerability. We use this information as input for SZZ to identify the set of inducing commits (Step 3). As output we have $C_{r_i} = \{ic_1, ic_2, ic_3, ic_4\}$).

In this example, we consider ic_4 as the turning point as it is the last inducing commit. It touched the files $F = \{A, B, D\}$. We consider that the

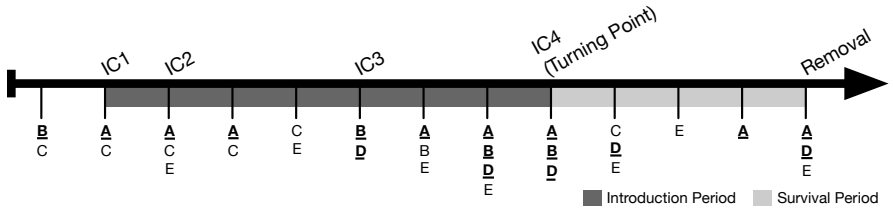


Fig. 1: Example of a vulnerability's life-cycle. IC = Inducing Commit; {A, B, C, D, E} = files of the project (underlined and bolded if changed by the commit).

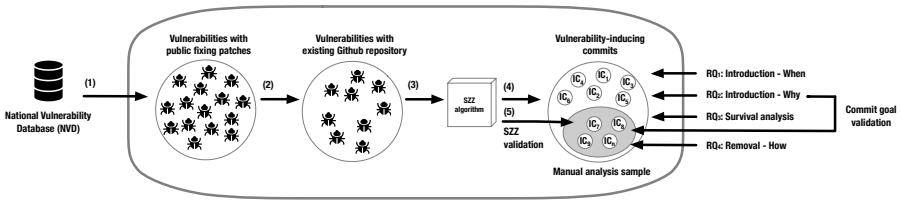


Fig. 2: Process to extract data to answer our research questions.

vulnerability required file A to change 1 (ic_1), 2 (ic_2), 2 (ic_3) and 6 (ic_{x4}) commits. As such, we compute a minimum of 1 commit, a maximum of 6 commits, an average of 2.75 commits, and a median of 2 commits on file A for the vulnerability be introduced. We compute these statistics for every ic_i and every f_j.

Furthermore, we analyze the survivability of the vulnerability, For this, we compute the number of commits for each f_j and the days spent until the vulnerability's removal. For instance, in this example, two commits touched A until the vulnerability was removed.

2.5 Data Extraction

Figure 2 presents an overview of our data extraction process. For the sake of comprehensibility of the overall methodology employed, in Section 2.4 we present an example of a vulnerability's life-cycle that will help the reader understand better the extraction process. First, we mine vulnerabilities from NVD. Rather than develop our own mining tool, we exploit CVE-Search [3], an open-source tool that imports the entire set of CVEs of the NVD repository into a MongoDB database that facilitates CVE search and processing. To study the life-cycle of these extracted issues, we need a set of vulnerabilities having a well-defined introduction and fixing time. For this reason, we filter the vulnerabilities with fixing patches $p_i \ \epsilon \ C_{r_i}$. (Step 1 in Figure 2). As a result, we have a set V with 2,973 vulnerabilities from projects using

different programming languages, such as C and Java. To investigate the lifecycle of the vulnerabilities, we mine the repositories of the projects in which they appear. Therefore, for each vulnerability v_i, r_i must still be available on GitHub, once through this platform we have access to r_i's history. We remove the vulnerabilities that do not have this property from V (Step 2 in Figure 2).

Subsequently, for each vulnerability v_i, we identify the introducing commit ic_i (also, *vulnerability-inducing commit*) by using the SZZ algorithm [79] (Step 3 in Figure 2). More specifically, given a vulnerability-fixing commit (p_i), the algorithm returns the set of commits that last changed the files touched by the commit. In summary, for each f_i in the input commit, SZZ (1) obtains the diff with respect to the previous commit, (2) retrieves the list of deleted lines, and (3) blames the file and obtains the commits where those lines were changed last. For each v_i, we run SZZ with p_i as input and obtain the set of commits that likely introduced v_i (a vulnerability can be introduced in one or more commits). It found vulnerability-inducing commits for 1,195 vulnerabilities, of which 651 require more than one commit to appear. We focus on this set of vulnerabilities, while we exclude the remaining 1,778 that do not have complete information.

2.6 Data Validation

As recent work has reported the SZZ algorithm to be poorly accurate [34, 73]; we empirically verify the performance of SZZ on our dataset. Two of the authors of this paper (the *inspectors*) independently analyze a statistically significant set of 307 vulnerability-inducing commits (confidence interval=0.95). In doing so, they rely on the source code of both vulnerability-fixing and vulnerability-inducing commits: the task is performed to understand whether the change applied in the vulnerability-inducing version actually introduced the vulnerability that is fixed in the vulnerability-fixing version. After the independent inspection, the two inspectors compared their assessments, finding an agreement in 97% of the cases; they discussed the remaining 3% until an agreement was found. All in all, the precision of the SZZ algorithm was 91% (280 correct vulnerability-inducing commits): this result strongly differs from previous findings [73] and suggests that the performance of SZZ may depend on both the dataset exploited and the type of defects considered, i.e., general defect vs vulnerability.

2.7 **RQ$_1$.** When - Research Methodology

To understand *when* vulnerabilities are introduced, we proceed in a similar way as done in previous work [83]. Specifically, for each vulnerability-inducing commit, we first compute the number of commits performed on the source code file f_j where v_i occurs since the first commit involving f_j up to ic_i (Step 4 in Figure 2). In cases where a vulnerability is associated to multiple vulnerability-inducing commits, i.e., the SZZ algorithm gives more the one

commit as responsible for the introduction of v_i, we consider the files modified by the last inducing commit to compute minimum, maximum, average, and median number of commits (considering each inducing commit) required by v_i to gradually affect f_j. This way, we are able to verify whether vulnerabilities affect source code files since their introduction or if they are the effect of multiple maintenance and evolution operations performed by developers.

2.8 **RQ$_2$.** Circumstances - Research Methodology

To understand *under which circumstances* developers introduce vulnerabilities, we automatically classify each ic_i according to one or more of the categories described by Tufano et al. [83] (see Table 3). The categories include (i) the commit goals, i.e., the task the developer was performing when introduced a vulnerability, (ii) the project status, i.e., the development phase of the project, and (iii) the developer status, i.e., the characteristics of the developer who introduced the vulnerability. We automatically assign one or more of the categories to each ic_i as follows (Step 5 of Figure 2):

Commit goal. We rely on a previously proposed approach that analyzes the content of the commit messages and uses keyword-matching [42, 45, 67, 87]. As an example, a commit is classified as *'bug fixing'* if the corresponding commit message contains keywords like 'bug', 'defect', 'fix', 'repair', 'error', 'issue'. Our appendix contains the complete list of keywords used in this step [13]. Although the classification accuracy of this textual approach has been previously empirically evaluated [67], we proceed with a manual re-assessment of a statistically significant sample of vulnerability-inducing commits to evaluate its accuracy in our context. To this aim, we take into account the same set of 307 commits previously used to evaluate the SZZ accuracy (see Section 2.6): two authors of this paper independently analyzed the categories assigned by the keyword-matching approach. After the first phase, the agreement between the inspectors was 94%. After discussion, the accuracy of the textual approach was found to be 86%, confirming the previous findings [67].

Project status. Concerning this category, we compute the number of days from the inducing commit's date to the date of the nearest minor or major release and assigned the proper *'working on release'* category based on this number. A similar procedure is followed for the *'project startup'* category, computing the number of days from the inducing commit's date to the date in which the project started (the date of the first commit).

Developer status. Finally, we assign the developer status considering two perspectives. First, we compute the workload of the developers who introduced the vulnerabilities (the authors of the inducing commits): similarly to previous work [82, 83], we use the number of commits made by developers as proxy metric for workload, i.e., the higher the number of commits the higher the workload of a developer in a given time window. Specifically, given a ic_i

Table 3: Tags assigned to vulnerabilities inducing commits.

Tags	Description	Values
	COMMIT GOAL TAGS	
Bug fixing	The commit aimed at fixing a bug	[true,false]
Enhancement	The commit aimed at implementing an enhancement in the system	[true,false]
New feature	The commit aimed at implementing a new feature in the system	[true,false]
Refactoring	The commit aimed at performing refactoring operations	[true,false]
	PROJECT STATUS TAGS	
Working on release	The commit was performed [value] before the issuing of a major release	[one day/week/month, more than one month]
Project startup	The commit was performed [value] after the starting of the project	[one week/month/year, more than one year]
	DEVELOPER STATUS TAGS	
Workload	The developer had a [value] workload when the commit has been performed	[low,medium,high]
Tenure	The developer tenure when the commit was performed	[newcomer,medium, expert]

performed by a developer d during a month m, we first compute the workload distribution for all developers of the project during m. Then, we consider the workload of d to be 'low' if his/her number of commits is strictly lower than the first quartile of the distribution, 'medium' if between first and third quartile, and 'high' when higher than the third quartile. Secondly, we compute the *'project tenure'* category [68,85], that is, an experience metric that counts the number of months since the developer's first event on the project where the vulnerability was introduced. Starting from the distribution of all developers' project tenure, we consider d a 'newcomer' if his/her project tenure is strictly below the first quartile of the distribution and 'expert' if strictly higher than the third quartile; otherwise, we assign 'medium'.

2.9 $\mathbf{RQ_3}$. Survivability - Research Methodology

The inducing and the fixing commits allow us to investigate each vulnerability's survivability. In the case of vulnerabilities with more than one inducing commit, we only consider their last inducing commits, as they represent the turning point of the introduction process. Furthermore, given a vulnerability v_i, we define the time interval between its inducing commit ic_i and its fixing patch p_i as a *vulnerable interval* and determines the longevity of v_i. We can use a vulnerability longevity to compute the number of days between its introduction (ic_i.date) and its fixing patch (p_i.date), as well as the number of commits between ic_i and p_i that modified the file f_i where the vulnerability occurs

(Step 6 in Figure 2). These metrics complement each other: For example, two commits may occur in a short interval of days, but with a high number of commits in-between, therefore analyzing only the date of the commits would be misleading.

After collecting the data, we perform a survival analysis [59], a statistical method that aims at analyzing and modeling the time duration until one or more events happen. The survival function $S(t) = P\pi(T > t)$ indicates the probability that a vulnerability survives longer than a time t. The survival function does not increase as t increases; also, it is assumed that $S(0) = 1$ at the beginning of the observation period, and, for time $t \to \infty$, $S(\infty) \to 0$. The survival analysis aims at estimating a survival function from data and assessing the relationship of explanatory variables to survival time. In the context of our study, the population is represented by the vulnerabilities instances while the event of interest is its fix. Then, the 'time-to-death' of a vulnerability is represented by the observed time from its introduction to its fix. We refer to this time as the 'lifetime' of a vulnerability.

2.10 **RQ$_4$.** Removal - Research Methodology

To understand how vulnerabilities are removed from the source code (Step 7 of Figure 2), we perform a manual analysis of the patches corresponding to the statistically significant sample of 307 vulnerability-inducing commits identified when validating the data extraction process (see Section 2.6).

We followed an open coding process [47]: The 307 vulnerability fixes were equally distributed between two authors of the paper, who independently and manually categorized the kinds of actions performed by developers (e.g., a refactoring) relying on vulnerability-fixing commit message and the Git diff (Step 8 of Figure 2). Afterwards, the authors opened a discussion aimed at clarifying and standardizing the analysis process, which led to the joint re-analysis of all classifications made so far. The final outcome of this process consisted in the definition of a set of categories that explain how vulnerabilities are removed from the code.

2.11 Threats to Validity

In this section, we discuss and explain possible threats to the validity of our empirical study.

Construct validity. Our results can be affected by the wrong identification of both vulnerability fixing patches and vulnerability-inducing commits. As for the former, we relied on the vulnerability fixing patches available in NVD: while this database is curated and improved constantly, we cannot exclude that a patch may not remove the vulnerability as intended. As for the latter, we rely on the SZZ algorithm, which analyzes the previous history of a fixed file to identify the commit that likely introduced the vulnerability.

Previous studies have shown that this algorithm may frequently fail in the identification of the correct defect-inducing commit [34, 73]; to account for this aspect and control the reliability of SZZ in our context, we have manually re-assessed its performance on our dataset for a sample of 307 vulnerability-inducing commits, finding an accuracy of 91%. Although the metrics to compute the developers' workload has been used in the past [68, 82, 83], it is only an approximation, because different commits may require a different amount of work and a developer may work on various projects.

Internal validity. To address $\mathbf{RQ}_2$, we implemented an automatic script to automatically characterize vulnerability-inducing commits. To study the *'commit goal'* category, we re-implemented a previously proposed keyword-matching approach [42, 45, 67]: while it has been shown to be accurate [67], we further assessed its accuracy on our dataset by performing a manual analysis of the categories assigned to a statistically significant sample of vulnerability-inducing commits. Such an analysis confirmed previous accuracy. Moreover, the manual analysis performed did not reveal the existence of categories other than those proposed by Tufano et al. [83]. As for the other categories, we re-implemented previously proposed algorithms [68, 83, 85] following the exact description reported in those papers; despite the extensive testing phase, we cannot exclude possible implementation errors. For the sake of verifiability and replicability, we make all data/scripts used in this study publicly available [13].

Conclusion validity. The conclusions made with respect to the survivability of vulnerabilities ($\mathbf{RQ}_3$) rely on a statistical technique called survival analysis [59]. While this is a standard approach for investigating lifetime expectancy [43, 44], a possible threat concerns the metrics we used to determine the vulnerability survival, i.e., the number of commits and number of days from their introduction to removal. Project activity or developers' availability may influence the two variables. With respect to $\mathbf{RQ}_4$, we conducted a manual investigation to determine how vulnerabilities are removed: more authors of this paper have been involved in the process in order to increase the accuracy of the classification of vulnerability fixing strategies. Despite the final discussion among the authors, we cannot exclude imprecision and some degree of subjectiveness. Replications of our work are, therefore, still desirable.

External Validity: We took into account 1,195 of 62 different types coming from 616 open-source projects. As such, the size of our dataset is comparable with the one of other large-scale software engineering studies (e.g., [50, 83]). However, the life-cycle of vulnerabilities affecting systems developed in closed-source systems may differ. Thus, replications of our study in different settings would be desirable.

3 Analysis of Results

In this section, we present and discuss the results to our research questions that aimed to understand how and under which conditions vulnerabilities are introduced, their survivability and how they are removed from code.

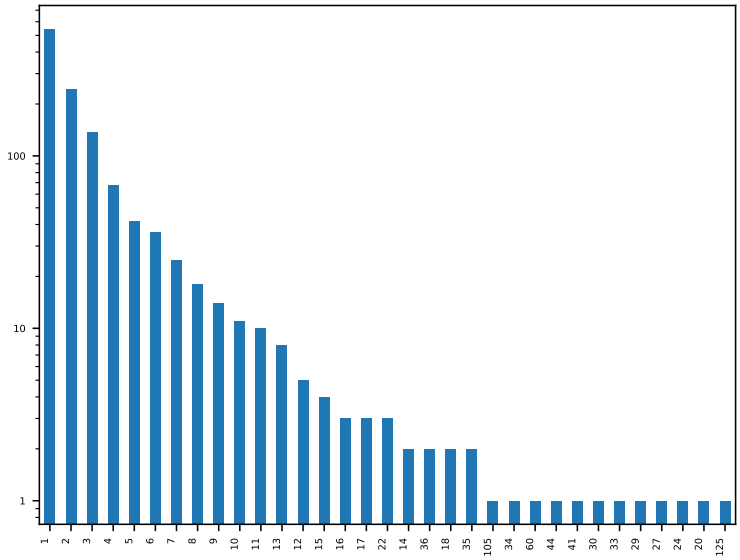


Fig. 3: Frequency of inducing commits per vulnerability in logarithmic scale.

RQ$_1$: *When are vulnerabilities introduced: during a new file creation or during activities on preexisting files?*

To address our first research question, we exploited SZZ to identify the one or more inducing commits of each of the 1,195 considered vulnerabilities. On total, the algorithm found 3,974 inducing commits: this means that, on average, each vulnerability requires 3.32 to manifest itself (median=2). Figure 3 shows an overview of the number of commits required to vulnerabilities appear in code. Based on these findings, we can argue that vulnerabilities are generally introduced in source code after a few commits, thus not necessarily requiring a large amount of evolutionary activities. Moreover, we found that 651 vulnerabilities, i.e., 55% of the total ones, have more than one inducing commit. The vulnerability with the highest number of inducing commits (125) is an SQL Injection in the Navigate CMS project (CVE-2018-17552), which was gradually introduced because of the large amount of modifications done by developers to the `login.php` module and, in particular, to the way external users can access the application. These continuous modifications likely had the effect of focusing on the functionality rather than properly checking for anomalous inputs, leading to the introduction of vulnerable code.

As part of our analysis, we also focus on the *affected files perspective*, meaning that we were interested in understanding how many commits touching a

certain file are required to introduce a vulnerability. To this aim, for each affected file we computed average and median number of commits between its creation and the introduction of a vulnerability in it, just considering the commits touching it. As a result, we identified a total of 4,343 touched files: on average, the number of files modified per inducing commit is 3.63 (median=1), while in 81 cases the inducing commits modified more than five files. This indicates that most of the times vulnerabilities are introduced in commits where developers apply changes on a limited set of files. On the one hand, this is somehow in contrast with the findings of Hindle et al. [46], who showed that large commits (involving more that 10 files) are instead more prone to introduce defects. On the other hand, our findings suggest that vulnerabilities are different from non-security defects, as also pointed out by Morrison et al. [61].

Looking deeper to the results of our analysis, we found that 1,980 files (46%) were created by an inducing commit, i.e., almost half of the files are created in a commit that introduces or gradually leads to the introduction of a vulnerability, and, more in general, the median number of commits required to a file to be affected by a vulnerability is 3. Also, we found three cases where a file was created during a unique inducing commit and that, therefore, born vulnerable in the first place. In general, these results tell us that, while creating new files, developers should be better supported by automated solutions aimed at pinpointing the presence of possible vulnerable code fragments. On the other hand, there is a non-negligible number of cases where the vulnerabilities appear after several evolutionary activities; indeed, 570 files (13%) require more than 500 commits to become vulnerable. Figure 4 shows the box plot of the number of commits that touched the files up to the inducing commits. The worst case observed was the one of the WordPress project, where a cross-site scripting vulnerability (CVE-2018-5776) was introduced in the `version.php` file after 11,511 commits on it. This result indicates that monitoring mechanisms should be put in place during the entire life-cycle of a project, despite the likelihood of introducing a vulnerability is much lower after the creation of a file.

RQ$_2$: *Under which circumstances are the vulnerabilities introduced?*

Still working on SZZ output, we investigated the circumstances in which vulnerabilities are introduced in source code. We classified the vulnerability inducing commits regarding their goals, project status, and developer status. Table 4 reports the percentage of vulnerability inducing commits assigned to each category. A commit may have several goals as we look the commits messages to match one or more of the following categories: *bug fixing*, *enhancement*, *new feature*, and *refactoring*. Overall, we classified 1,045 vulnerabilities commits with 2,298 commit goal categories, while the messages of the remaining 246 inducing commits did not match any of our categories: a missing categorization was due, for instance, to empty commit messages like in the case of a *cross-site scripting* vulnerability (CVE-2017-12474) of the Bento4 project [1].

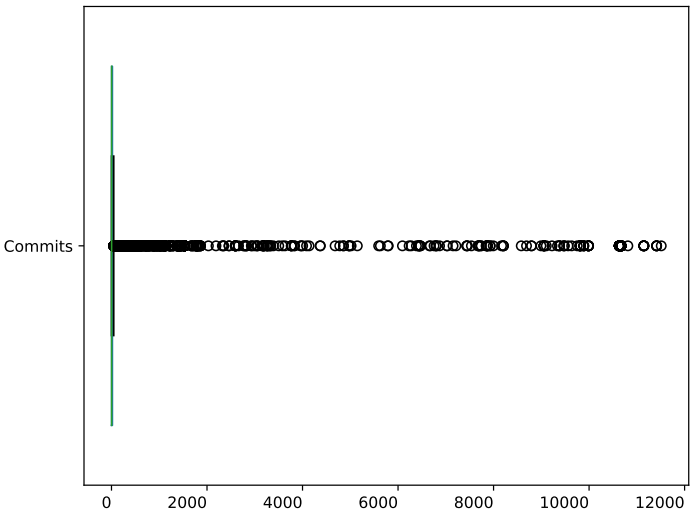


Fig. 4: Number of commits that touched the files up to the inducing commits.

Table 4: Categories of vulnerabilities inducing commits.

Commit Goal	New Feature	30%
	Bug Fixing	27%
	Enhancement	24%
	Refactoring	19%
Working on release	One day	67%
	One week	26%
	One month	6%
	More than one month	1%
Project Startup	One week	5%
	One month	1%
	One year	11%
	More than one year	83%
Workload	High	82%
	Medium	14%
	Low	4%
Tenure	Expert	94%
	Medium	5%
	Newcomer	1%

Summary for **RQ$_1$**

Source code files become affected by vulnerabilities just after 2 commits, on median, meaning that developers are more prone to introduce vulnerabilities during the initial activities on a newly created file. As such, vulnerability detection strategies should provide a closer support during the first development phases of new files.

We found that most inducing commits introduce a *new feature* (30%)—we classified 689 commits in this category. For example, we found an *out-of-bounds* vulnerability (CVE-2013-7456) in the LibGD project [8] with an inducing commit having the following message: *"add new interpolation method affine methods scale and rotation"*. In this case, we matched *add* in the *new feature* category. Furthermore, 614 (27%) inducing commits were classified as *bug fixing* activities. For instance, an *input validation* vulnerability (CVE-2010-4335) in the CakePHP project [2] was introduced by a commit that re-implemented form hashing security to use string-based keying. The commit involved five files, added 585 lines of code and deleted 686. Thus, our findings suggest that developers may introduce a security-related problem in the source code during maintenance activities, such as bug fixing, especially when the activity is complex and requires changes to several code entities and/or lines of code: this is in line with previous findings achieved when understanding how maintainability issues are introduced in open-source projects [83, 87]. Furthermore, we observed that 550 (24%) of the inducing commits are related to enhancement activities. More surprisingly, *refactoring* operations pertain to the introduction of 445 vulnerabilities (19%): this means that, while trying to improve source code quality, developers often fall into error and introduce security-related problems. On the one hand, this confirms previous findings on the hardness to apply refactoring [25, 65, 88]; on the other hand, our findings indicate that besides maintainability problems [26], the side effect of refactoring includes security issues. This is something that would deserve further investigations.

As mentioned before, a vulnerability could match more than one category. In this respect, we categorized 681 vulnerabilities with more than one tag. Overall, 184 vulnerabilities matched all categories. For instance, a *cross-site script* vulnerability (CVE-2017-15278) of the TeamPass project [16] has 20 inducing commits, we found the following messages among them: (i) *"Improved install/upgrade. Continuing code review"*; (ii) *"Fixed an issue with DUOSecurity login. Small improvement on searching on Items page. Added the possibility to search by Tag only. Several small fixes"*; and, (iii) *"Moved import and export features from Home to Items page. - Fixed bug on folders access (list to restreint)"*. We matched the words *upgrade*, *fixed*, *added* and *moved* to *enhancement*, *bug fixing*, *new feature* and *refactoring*, respectively. On average, we matched a vulnerability to 1.78 categories (median=2).

Besides the commit goal tags, we discovered that most of the vulnerabilities are introduced the last day before issuing a release. Indeed, the percentage of vulnerabilities introduced more than one month prior to issuing a release is

really low (1%). Additionally, most of the vulnerabilities are introduced after more than one year of the project startup (83%), while only 6% are introduced in projects newer than a month. Next, we looked at the developer status in the moment they introduced vulnerabilities. We found that the majority of developers have a high workload when performing vulnerability inducing commits (82%), while 14% and 4% of them have medium and low workloads, respectively. Regarding the tenure of developers, we found that most vulnerabilities are introduced by experts (94%), again confirming findings reported in literature on the source code drawbacks introduced by expert developers [36, 82]. Furthermore, we found that most vulnerabilities (83%) are introduced in advanced phases of the development process and in most of the cases the day before a deadline (67%). These results may be due to the fact that developers have high workload during these periods, which complies with our findings that most developers (82%) have high workload when performing the vulnerability inducing commits. On the other side, developers who introduced the vulnerabilities are experts, which may be due to the fact that more experienced developers tend to perform more complex and critical tasks [89]. As such, our findings suggest the need for improved methods on how to schedule resources in software projects able to take into account the potential security drawbacks that developers may introduce.

Summary for **RQ$_2$**

Developers mostly introduce vulnerabilities when introducing new features in the source code, but in a non-negligible number of cases refactoring is the main cause for that. Moreover, the majority of the vulnerabilities are introduced up to a month before a deadline, and after the first year from the project startup. Finally, expert developers with high workloads are more prone to introduce vulnerabilities.

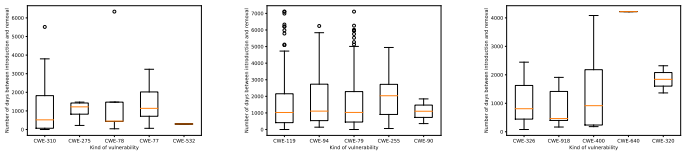
RQ$_3$: *What is the survivability of vulnerabilities?*

Results: We analyzed the vulnerabilities regarding the interval between their last vulnerability inducing commits and their fixing patches. We considered on the last inducing commits as they are the turning point where the code becomes vulnerable. Figure 5 shows the box plot of the distribution of the number of days needed to fix some of the kinds of vulnerabilities. Figure 6 shows the box plot of the distribution of the number of commits needed to fix some of the kinds of vulnerabilities The remaining box plots are also available online [13].

Table 5 shows the descriptive statistics of the survival distribution of the number of commits days when aggregating all vulnerability kinds considered in our study. In particular, the median value of such distributions is 1,043 and 1,546 for days and commits, respectively. We found a *double free* vulnerability

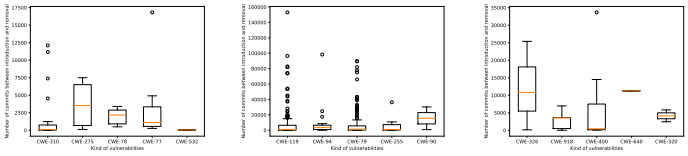
Table 5: Statistics of the number of commits and days needed for a vulnerability in the system before being removed.

Survival	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
Days	0	410.8	1,043	1,535.1	2,238.8	7912
Commits	2	298.2	1,546.5	7,082.8	6111	351,160



(a) (b) (c)

Fig. 5: Survivability of vulnerabilities in terms of days.



(a) (b) (c)

Fig. 6: Survivability of vulnerabilities in terms of commits.

(high severity) in the code of the *krb5* project (CVE-2017-11462) with 17 inducing commits, the last one occurred on 1995. This vulnerability was fixed in 2017, 7,912 days later. In total, 23 (1.2%) vulnerabilities were introduced and fixed the same day. Moreover, an *out-of-bounds write* vulnerability of the *LibreOffice* [9] project (CVE-2017-7870) took the highest number of commits to be fixed (351,160). In total, 17 (1.1%) vulnerabilities were fixed after only two commits, most of them the same day they were introduced.

Discussion: For each vulnerability, we analyzed the interval delimited by its last vulnerability inducing commit and its fixing patch (removing commit). Figure 5 shows the box plot of the distribution of the number of days needed to fix some vulnerabilities, the box plots of the remaining vulnerabilities are available online [13]. The box plots, depicted in log-scale, show that most vulnerabilities are fixed after a long period of time (i.e., over 500 days). Figure 6 shows the box plot of the distribution of the number of commits needed to fix some of the kinds of vulnerabilities. The box plots of the remaining kinds are available online [13]. Notice that some kinds of vulnerabilities are removed after a small number of commits, while others are fixed after thousands commits. On

median, aggregating all kinds of vulnerabilities, developers took 1,546 commits to fix the vulnerabilities.

The vulnerability that took more days (7,912) to be fixed was a double free vulnerability on the Kerberos project [7] (CVE-2017-11462). Developers performed 13,415 commits between its introduction and its fix. On the other hand, the vulnerability that took the highest number of commits (351,160) to be fixed was a *out-of-bounds write* in the LibreOffice project [9] (CVE-2017-7870). It took 5,029 days for this vulnerability to be removed. Its last inducing commit was performed on March/2003 and its fixing commit on January 2017 by a different developer. This vulnerability represents the biggest difference between the survival days and survival commits of a vulnerability. A *cross-site scripting* of the OAuth2orize project [12] (CVE-2018-11647) took both six days and commits from the last inducing commit to the fixing commit (performed by different developers). Moreover, a *cross-site scripting* vulnerability of the PHPmyadmin project [14] (CVE-2011-3592) took only 148 days to be fixed. However, 20,517 commits were performed before the fix.

Following, we found that 591 (39.9%) vulnerabilities were introduced (last inducing commits) and removed by the same developer of 303 projects. Among these vulnerabilities, 36 occurred in the ImageMagick project [6] and a unique developer is responsible for both introducing and removing 34 of them from September/2009 until August/2018. The same developer introduced 40 among 45 vulnerabilities belonging to this project, and has the highest rate of inserting vulnerabilities in the code. A developer of the TCPdump project [15] project has the highest rate of vulnerability removal (48).

Summary for **RQ$_3$**

In general, vulnerabilities survive in code for a large number of days and commits. In addition, 39% of the vulnerabilities are both introduced and removed by the same developer.

RQ$_4$: *How are vulnerabilities removed?*

To address our last research question, we randomly analyzed 307 vulnerabilities of 43 CWE belonging to 213 projects (95% confidence level and a 5% confidence interval). We analyzed their fixing commit messages and changed lines of code. We found that each vulnerability may be removed in several ways due to different programming languages and/or to the developer's preferences: for this reason, we could not come up with a taxonomy reporting the recurrent patterns followed by developers when fixing most of the vulnerabilities. Nevertheless, in our analysis we were able to identify and classify recurrent methods adopted when removing five specific kinds of vulnerabilities. Table 6 presents an overview of our results.

Table 6: Methods used to remove vulnerabilities.

CWE	Kind	Removal Method	Total
CWE-79	Improper Neutralization of Input During Web Page Generation (Cross-site Scripting)	Use Escape functions for HTML entities	31%
		Sanitize the text of user inputs	15%
		Manual control of strings to prevent them from containing HTML or PHP elements	10%
CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	Add controls on the buffer size	62%
		Cast between different types of variables containing the buffer size	12%
CWE-264	Permissions, Privileges, and Access Controls	Hide ids within forms	20%
		Check that only authenticated users can access some functions	13%
		Check that authenticated users cannot access sensitive project data	13%
CWE-89	Improper Neutralization of Special Elements used in an SQL Command (SQL Injection)	Control on query parameters to prevent them from containing SQL code	21%
		Queries are parameterized	14%
CWE-352	Cross-Site Request Forgery (CSRF)	Checks to verify if the request is valid	33%
		Added token to identify users	25%

In the first place, we noticed that most fixes involve multiple files (57%). Indeed, the number of involved files was 2 on median, with 409 being the worst case. Moreover, developers added and deleted on median 11 and 4 lines, respectively. In all fixes at least a line of code was deleted, while three fixes did not add any line in code. The highest number of added and deleted lines was 27,911 and 4,686, respectively.

Looking deeper at the manual classification done, the five kinds of vulnerabilities which developers removed with recurrent methods are (i) *improper neutralization of input during web page generation* (CWE-79) with 65 occurrences; (ii) *improper restriction of operations within the bounds of a memory buffer* (CWE-119) with 34 occurrences; (iii) *permissions, privileges, and access controls* (CWE-264) with 15 occurrences; (iv) *improper neutralization of special elements used in an SQL command* (CWE-89) with 14 occurrences; and (v) *cross-site request forgery* (CWE-352) with 12 occurrences. It is important to note that these kinds are part of the top 10 frequent vulnerabilities in our study (see Table 1) and, perhaps more importantly, three of them appear to be in the top-25 most dangerous software errors according to CWE.[1]

Looking at Table 6, we can notice that most of the times developers use to apply a few, specific operations to remove the considered vulnerabilities. For example, the *improper neutralization of input during web page generation* is removed by escaping functions in 31% of the cases, while sanitizing the text of user inputs (15%) and manually checking strings (10%) are less popular, yet recurrent operations to solve this issue. Similar conclusions can be drawn when

[1] https://cwe.mitre.org/top25/index.html

considering the other vulnerability types. More interestingly, we can highlight that the fixes often involve simple program transformations: a clear example is represented by the *add control on the buffer size* operation, which is a widely used method to remove out of memory buffers (it is used in 62% of the cases). This program transformation just requires the addition of an `if` statement that restricts the bounds of an input. Similarly, most of the other operations usually performed by developers relate to simple actions that could, potentially, be automated and used in combination with vulnerability detection approaches to cover the entire identification/refactoring pipeline and thus substantially support developers when dealing with security-related issues.

Summary for **RQ$_4$**

The most harmful vulnerability types can be removed through simple program transformations that involve the addition of functions or checks in the code, even when large number of files and lines of code are involved.

4 Discussion and Implications

The results of our four research questions provided a number of insights that need to be further discussed as well as several implications for the software engineering research community. Specifically:

The first phase effect. According to our findings, vulnerabilities are often introduced shortly after the creation of new files (on median, after 2 commits) and while introducing new features in a software project. On the one hand, our findings contradict the results of previous studies where researchers have showed that software reliability issues, and in particular vulnerabilities, are mainly related to evolutionary activities done by developers [77, 84], thus stimulating further research on the origin of software issues [74]. On the other hand, they have clear implications for future research on automated solutions aimed at finding vulnerabilities in source code: indeed, we argue that novel techniques able to be *time- and context-dependent*, i.e., analyzing the timeline as well as the intended actions of developers of newly committed source code files, should be devised and experimented.

More research on vulnerabilities is needed. Our study provided the first, large-scale analysis of different aspects of the life-cycle of software vulnerabilities. Nevertheless, we still have no compelling empirical evidences of the reasons why they remain in source code for large periods of time, e.g., because developers are not aware of their presence or because they cannot realize how to remove them. Similarly, we are still unaware of the motivations why certain actions, e.g., new feature implementation, are more prone

to introduce vulnerabilities: for example, this may be due to the lack of proper control mechanisms or a lack of knowledge of basic security principles. All these aspects represent challenges for the empirical software engineering research community and our study stimulates a more comprehensive investigation into software vulnerabilities and security practices in general.

On refactoring effects on software reliability. As shown in our study, in a non-negligible number of cases developers introduce vulnerabilities when doing refactoring. This is a pretty controversial finding: indeed, this activity is supposed to improve source code quality, while sometimes it leads to the introduction of both software quality and reliability problems [26]. Likely, our results represent a reflection of what has been shown in literature, namely that developers do not have proper mechanisms that allow the automation of refactoring activities [87] and that, therefore, apply them manually [27, 65]. In any case, our findings uncover another potential issue of a wrong application of refactoring, i.e., the introduction of vulnerabilities. We argue that the effect of refactoring activities on software reliability should be further investigated by the research community.

On security testing and more. Our results indicate that several vulnerabilities are introduced by developers close to deadlines and, perhaps more importantly, remain in a system for a long while. We argue that such results are important for two main aspects. On the one hand, the definition of advanced testing mechanisms able to assist developers when looking for possible security issues should be devised: this represents a further challenge for the entire testing community and, perhaps, even for automatic test case generation [40, 69, 76]. On the other hand, enabling alternative strategies, e.g., code review practices for vulnerabilities [35], could be an interesting path to allow developers in better spotting errors statically [23, 71].

Scheduling resources better. One of the key results of our investigation highlighted that expert developers are those that tend to introduce vulnerabilities more. This may likely be due to their high workload or to the pressure they have to respect upcoming deadlines. These results call for a more comprehensive overview of software vulnerability research: for instance, novel methods to triage them by optimizing expertise and workload could nicely complement the current research on the topic. Similarly, it is still unknown how more general social issues among developers, e.g., social debt [31,68,81], influence the introduction and/or development of security issues.

Automating vulnerability removal. Besides the opportunities for improving vulnerability identification mechanisms, one of the main results of our empirical study reveal some important insights into the removal of software vulnerabilities. Indeed, from the last research question we discovered that most of the actions done by developers to remove vulnerabilities are simple program transformations (e.g., escaping functions) that can be automated and made available to developers in order to better support the removal of vulnerabilities as well as to implement a comprehensive pipeline covering their entire life-cycle, from discovery to fixing.

5 Related Work

Morrison et al. [61] stated that if vulnerabilities behave in the same way as non-security defects in code, then existing defect prevention methods may be used unchanged. Otherwise, understanding how they differ could inform software development process improvement for security. They extended the Orthogonal Defect Classification [32], a scheme software development teams use to collect and analyze defect data to aid software development process improvement, to study process related differences between vulnerabilities and defects, creating ODC + Vulnerabilities (ODC+V). They applied it to classify 583 vulnerabilities and 583 defects across 133 releases of Firefox, phpMyAdmin, and Chrome, and found that they behave differently. Our study complements Morrison et al.'s work [61], and our findings can be used to improve defect preventions methods.

Shahzad et al. [77] conducted an exploratory measurement study to investigate the life-cycle of 46,310 vulnerabilities. They investigated the phases in the life-cycle, the evolution of vulnerabilities during the years, the requirements to exploit them, their functionality and risk level, and the software vendors and products. They consider that the life-cycle of a vulnerability starts when it is discovered by the vendor, a hacker, or any third-party software analyst. It ends when all users of the software install the vulnerability fixing patch. In this work, we also investigate the life-cycle of vulnerabilities. However, we consider a life-cycle the period between the vulnerability introduction in the source code and its removal from the code. This way, we investigate aspects that may help developers during the development process by understanding how these problems are introduced and removed in practice.

Smith et al. [80] investigated how developers use a security-focused statistic analysis to resolve security defects. They conducted an exploratory think-aloud study with ten developers who had contributed to a security-critical medical records software system written in Java. During the study, they observed each developer as they assessed potential security vulnerabilities identified by Find Security Bugs [4], a static analysis tool to help developers remove security defects early in the development life-cycle. They found that developers ask questions not only about security vulnerabilities, associated attacks, and fixes, but also questions about the software itself, the social ecosystem that built the software, and related resource and tools. Complementing their work [80], in this paper, we study in which circumstances developers introduce them by analyzing the commits' goals and the actions developers took on the code to remove these vulnerabilities. Together with Smith et al. [80] findings, our analysis results can be used to help creating useful tools to detect security problems, and/or to improve existing ones.

Previous work [22, 49, 54, 55] have evaluated security tools that aim at helping developers find and remove vulnerabilities. Mostly, they evaluate the number of vulnerabilities and false positives the tools report. For instance, Austin and Williams [22] compared the effectiveness of four existing techniques for finding vulnerabilities: systematic manual penetration testing, exploratory

manual penetration testing, static analysis, and automated penetration testing. Unlike the previous studies, we studied the complete life-cycle of vulnerabilities and provided a large dataset of vulnerabilities data that allowed a better understanding of when and in which circumstances they occur, how long they stay in code and how developers remove them. Our insights and data may be useful to improve the security finding detection scenario.

Murphy-Hill et al. [64] studied refactoring practice at large. They studied data from a variety of data sets spanning more than 39,000 developers, 240,000 tool-assisted refactorings, 2,5000 developers hours, and 12,000 version control commits. Among their findings, they stated that refactorings are indeed performed and the kind or refactoring performed with a tool differs from the kind performed manually. However, they do not consider vulnerabilities in their study. In this study, we analyzed the life-cycle of 1,195 vulnerabilities, including the manual analysis of how they removed 307 vulnerabilities. In future work, our results can be used to propose refactoring tools to help developers when removing this kind of problems.

6 Conclusion

In this paper, we presented a large-scale empirical study conducted over the life-cycle of 1,195 of 62 kinds belonging to 616 open source projects. We aimed at understanding when and under which circumstances vulnerabilities are introduced, what is their survivability, and how they are removed. These results provide several valuable findings for the research community: (i) most of the times vulnerabilities are introduced after the first year of the project startup; (ii) developers introduce vulnerabilities as consequence of bug fixing activities; (iii) expert developers introduce most vulnerabilities, which is probably due to the complexity of their activities; and, (iv) vulnerabilities have high survivability rates regarding both the number of commits and of days they stay in code.

As future work, we plan to empirically compare and evaluate existing detectors in the literature to better understand the reasons why they may not help developers early detect vulnerabilities in the code, for example, they may be hard to use. Following, we aim at designing and developing a new generation of vulnerability detectors.

Acknowledgements Bacchelli, Palomba, and Braz gratefully acknowledge the support of the Swiss National Science Foundation through the SNF Projects No. PP00P2_170529 and PZ00P2_186090.

References

1. Bento4. https://github.com/axiomatic-systems/Bento4.
2. Cakephp. https://github.com/cakephp/cakephp.
3. Cve search tool. https://github.com/cve-search/cve-search.

4. Find security bugs. https://find-sec-bugs.github.io/.
5. Github. https://github.com/.
6. Imagemagick. https://github.com/ImageMagick/ImageMagick.
7. Kerberos. https://github.com/krb5/krb5.
8. Libgd. https://github.com/libgd/libgd.
9. Libreoffice. https://github.com/LibreOffice/core.
10. Mediacenter. https://www.pandasecurity.com/mediacenter/security/consequences-not-applying-patches/.
11. National vulnerability database. https://nvd.nist.gov/.
12. Oauth2orize. https://github.com/jaredhanson/oauth2orize-fprm.
13. Paper vulnerabilities database. TODO:site.
14. Phpmyadmin. https://github.com/phpmyadmin/phpmyadmin.
15. Tcpdump. https://github.com/the-tcpdump-group/tcpdump.
16. Teampass. https://github.com/nilsteampassnet/TeamPass.
17. U.s. nist computer security division. https://www.nist.gov.
18. Wannacry ransomware attack. https://en.wikipedia.org/wiki/WannaCry_ransomware_attack.
19. O. Alhazmi, Y. Malaiya, and I. Ray. Measuring, analyzing and predicting security vulnerabilities in software systems. *Computers & Security*, 26(3):219–228, 2007.
20. D. Aranha, P. Barbosa, T. Cardoso, C. Araújo, and P. Matias. The return of software vulnerabilities in the brazilian voting machine. *Computers & Security*, 2019.
21. S. Ardi, D. Byers, P. Meland, I. Tondel, and N. Shahmehri. How can the developer benefit from security modeling? In *Proceedings of the International Conference on Availability, Reliability and Security*, pages 1017–1025, 2007.
22. A. Austin and L. Williams. One technique is not enough: A comparison of vulnerability discovery techniques. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*, pages 97–106, 2011.
23. Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 2013 international conference on software engineering*, pages 712–721. IEEE Press, 2013.
24. N. Baddoo and T. Hall. De-motivators for software process improvement: an analysis of practitioners' views. *Journal of Systems and Software*, 66(1):23–33, 2003.
25. G. Bavota, A. De Lucia, M. Di Penta, R. Oliveto, and F. Palomba. An experimental investigation on the innate relationship between quality and refactoring. *Journal of Systems and Software*, 107:1–14, 2015.
26. Gabriele Bavota, Bernardino De Carluccio, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Orazio Strollo. When does a refactoring induce bugs? an empirical study. In *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*, pages 104–113. IEEE, 2012.
27. Gabriele Bavota, Andrea De Lucia, Andrian Marcus, Rocco Oliveto, and Fabio Palomba. Supporting extract class refactoring in eclipse: The aries project. In *Proceedings of the 34th International Conference on Software Engineering*, pages 1419–1422. IEEE Press, 2012.
28. G. Canfora and L. Cerulo. Impact analysis by mining software and change request repositories. In *Proceedings of the International Software Metrics Symposium*, pages 9–29, 2005.
29. M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, pages 147–160, 2006.
30. G. Catolino, F. Palomba, A. Zaidman, and F. Ferrucci. Not all bugs are the same: Understanding, characterizing, and classifying bug types. *Journal of Systems and Software*, 2019.
31. Gemma Catolino, Fabio Palomba, Damian A Tamburri, Alexander Serebrenik, and Filomena Ferrucci. Gender diversity and women in software teams: How do they affect community smells? In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Society*, pages 11–20. IEEE Press, 2019.
32. R. Chillarege, I. Bhandari, J. Chaar, M. Halliday, D. Moebus, B. Ray, and M. Wong. Orthogonal defect classification-a concept for in-process measurements. *Transactions on Software Engineering*, 18(11):943–956, 1992.

33. I. Chowdhury and M. Zulkernine. Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. *Journal of Systems Architecture*, 57(3):294 – 313, 2011.
34. D. Alencar da Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho, and A. Hassan. A framework for evaluating the results of the szz approach for identifying bug-introducing changes. *Transactions on Software Engineering*, 43(7):641–657, 2017.
35. M. di Biase, M. Bruntink, and A. Bacchelli. A security perspective on code review: The case of chromium. In *Proceedings of the International Working Conference on Source Code Analysis and Manipulation*, pages 21–30, 2016.
36. Dario Di Nucci, Fabio Palomba, Giuseppe De Rosa, Gabriele Bavota, Rocco Oliveto, and Andrea De Lucia. A developer centered bug prediction model. *IEEE Transactions on Software Engineering*, 44(1):5–24, 2017.
37. B. Du Bois, S. Demeyer, and J. Verelst. Refactoring-improving coupling and cohesion of existing code. In *Proceedings of the Working Conference on Reverse Engineering*, pages 144–151, 2004.
38. M. Finifter, D. Akhawe, and D. Wagner. An empirical study of vulnerability rewards programs. In *Proceedings of the USENIX Conference on Security*, pages 273–288, 2013.
39. M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.
40. Gordon Fraser and Andrea Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, 2012.
41. S. Frei, D. Schatzmann, B. Plattner, and B. Trammell. *Modeling the security ecosystem - the dynamics of (In)security*, pages 79–106. Springer US, 2010.
42. Y. Fu, M. Yan, X. Zhang, L. Xu, D. Yang, and J. Kymer. Automated classification of software change messages by semi-supervised latent dirichlet allocation. *Information and Software Technology*, 57:369–377, 2015.
43. P. Heagerty and Y. Zheng. Survival model predictive accuracy and roc curves. *Biometrics*, 61(1):92–105, 2005.
44. R. Henderson, M. Jones, and J. Stare. Accuracy of point predictions in survival analysis. *Statistics in Medicine*, 20(20):3083–3096, 2001.
45. A. Hindle, D. M German, M. Godfrey, and R. Holt. Automatic classication of large changes into maintenance categories. In *Proceedings of the International Conference on Program Comprehension*, pages 30–39, 2009.
46. A. Hindle, D. M German, and R. Holt. What do large commits tell us?: a taxonomical study of large commits. In *Proceedings of the International Working Conference on Mining Software Repositories*, pages 99–108, 2008.
47. H. Hsieh and S. Shannon. Three approaches to qualitative content analysis. *Qualitative health research*, 15(9):1277–1288, 2005.
48. S. Huang, H. Tang, M. Zhang, and J. Tian. Text clustering on national vulnerability database. In *Proceedings of the International Conference on Computer Engineering and Applications*, volume 2, pages 295–299, 2010.
49. N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: a static analysis tool for detecting web application vulnerabilities. In *Proceedings of the Symposium on Security and Privacy*, pages 258–263, 2006.
50. Y. Kamei, E. Shihab, B. Adams, A. Hassan, A. Mockus, A. Sinha, and N. Ubayashi. A large-scale empirical study of just-in-time quality assurance. *Transactions on Software Engineering*, 39(6):757–773, 2013.
51. B. Kitchenham. Evaluating software engineering methods and tool part 1: The evaluation context and evaluation methods. *Software Engineering Notes*, 21(1):11–14, 1996.
52. B. A Kitchenham, T. Dyba, and M. Jorgensen. Evidence-based software engineering. In *Proceedings of the international conference on software engineering*, pages 273–281, 2004.
53. F. Li and V. Paxson. A large-scale empirical study of security patches. In *Proceedings of the Conference on Computer and Communications Security*, pages 2201–2215, 2017.
54. V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the Conference on USENIX Security Symposium*, pages 18–18, 2005.
55. M. Martin, B. Livshits, and M. Lam. Finding application errors and security flaws using pql: A program query language. In *Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 365–383, 2005.

56. G=. McGraw. *Software Security: building security in*. Addison-Wesley, 2006.
57. P. Mell, K. Scarfone, and S. Romanosky. Common vulnerability scoring system. *IEEE Security & Privacy*, 4(6):85–89, 2006.
58. T. Menzies, A. Butcher, D. Cok, A. Marcus, L. Layman, F. Shull, B. Turhan, and T. Zimmermann. Local versus global lessons for defect prediction and effort estimation. *Transactions on Software Engineering*, 39(6):822–834, 2013.
59. R. Miller. *Survival Analysis*. John Wiley and Sons, 2011.
60. S. Mirhosseini and C. Parnin. Can automated pull requests encourage software developers to upgrade out-of-date dependencies? In *Proceedings of the International Conference on Automated Software Engineering*, pages 84–94, 2017.
61. P. Morrison, R. Pandita, X. Xiao, R. Chillarege, and L. Williams. Are vulnerabilities discovered and resolved like other defects? *Empirical Software Engineering*, 23(3):1383–1421, 2018.
62. P. Morrison, B. Smith, and L. Williams. Surveying security practice adherence in software development. In *Proceedings of the Hot Topics in Science of Security: Symposium and Bootcamp*, pages 85–94, 2017.
63. R. Muniz, L. Braz, R. Gheyi, W. Andrade, B. Fonseca, and M. Ribeiro. A qualitative analysis of variability weaknesses in configurable systems with #ifdefs. In *Proceedings of the International Workshop on Variability Modelling of Software-Intensive Systems*, pages 51–58, 2018.
64. E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. *Transactions on Software Engineering*, 38(1):5–18, 2012.
65. Emerson Murphy-Hill, Chris Parnin, and Andrew P Black. How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, 38(1):5–18, 2011.
66. S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller. Predicting vulnerable software components. In *Proceedings of the Conference on Computer and Communications Security*, pages 529–540, 2007.
67. F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, and A. De Lucia. The scent of a smell: An extensive comparison between textual and structural smells. *Transactions on Software Engineering*, 44(10):977–1000, 2018.
68. F. Palomba, D. Tamburri, F. Fontana, R. Oliveto, A. Zaidman, and A. Serebrenik. Beyond technical aspects: How do community smells influence the intensity of code smells? *Transactions on Software Engineering*, 2018.
69. Fabio Palomba, Annibale Panichella, Andy Zaidman, Rocco Oliveto, and Andrea De Lucia. Automatic test case generation: What if test code quality matters? In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 130–141. ACM, 2016.
70. L. Pascarella, F. Palomba, and A. Bacchelli. Fine-grained just-in-time defect prediction. *Journal of Systems and Software*, to appear, 2019.
71. Luca Pascarella, Davide Spadini, Fabio Palomba, Magiel Bruntink, and Alberto Bacchelli. Information needs in contemporary code review. *Proceedings of the ACM on Human-Computer Interaction*, 2(CSCW):135, 2018.
72. C. Pfleeger and S. Pfleeger. *Security in computing*. Prentice Hall Professional Technical Reference, 2002.
73. G. Rodríguez-Pérez, G. Robles, and J. González-Barahona. Reproducibility and credibility in empirical software engineering: A case study based on a systematic literature review of the use of the szz algorithm. *Information and Software Technology*, 99:164–176, 2018.
74. Gema Rodríguez-Pérez, Andy Zaidman, Alexander Serebrenik, Gregorio Robles, and Jesús M González-Barahona. What if a bug has a different origin?: Making sense of bugs without an explicit bug introducing change. In *Proceedings of the 12th ACM/IEEE international symposium on empirical software engineering and measurement*, page 52. ACM, 2018.
75. N. Sae-Lim, S. Hayashi, and M. Saeki. Context-based code smells prioritization for prefactoring. In *Proceedings of the International Conference on Program Comprehension*, pages 1–10, 2016.
76. Hossain Shahriar and Mohammad Zulkernine. Automatic testing of program security vulnerabilities. In *2009 33rd Annual IEEE International Computer Software and Applications Conference*, volume 2, pages 550–555. IEEE, 2009.

77. M. Shahzad, M. Z. Shafiq, and A. X. Liu. A large scale exploratory analysis of software vulnerability life cycles. In *Proceedings of the International Conference on Software Engineering*, pages 771–781, 2012.
78. R. Shirey. Internet security glossary. In *RFC*, 2000.
79. J. Sliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *Proceedings of the International Workshop on Mining Software Repositories*, pages 1–5, 2005.
80. J. Smith, B. Johnson, E. Murphy-Hill, B. Chu, and H. Richter. How developers diagnose potential security vulnerabilities with a static analysis tool. *Transactions on Software Engineering*, XX(XX):XX–XX, 2018.
81. Damian A Tamburri, Fabio Palomba, Alexander Serebrenik, and Andy Zaidman. Discovering community patterns in open-source: A systematic approach and its evaluation. *Empirical Software Engineering*, 24(3):1369–1417, 2019.
82. M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk. When and why your code starts to smell bad. In *Proceedings of the International Conference on Software Engineering-Volume 1*, pages 403–414, 2015.
83. M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk. When and why your code starts to smell bad (and whether the smells go away). *IEEE Transactions on Software Engineering*, 43(11):1063–1088, 2017.
84. Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. An empirical investigation into the nature of test smells. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 4–15. IEEE, 2016.
85. B. Vasilescu, D. Posnett, B. Ray, M. GJ van den Brand, A. Serebrenik, P. Devanbu, and V. Filkov. Gender and tenure diversity in github teams. In *Proceedings of the Conference on Human Factors in Computing Systems*, pages 3789–3798, 2015.
86. C. Vassallo, S. Panichella, F. Palomba, S. Proksch, A. Zaidman, and H. Gall. Context is king: The developer perspective on the usage of static analysis tools. In *Proceedings of the International Conference on Software Analysis, Evolution and Reengineering*, pages 38–49, 2018.
87. Carmine Vassallo, Giovanni Grano, Fabio Palomba, Harald C Gall, and Alberto Bacchelli. A large-scale empirical exploration on refactoring activities in open source software projects. *Science of Computer Programming*, 180:1–15, 2019.
88. Carmine Vassallo, Fabio Palomba, and Harald C Gall. Continuous refactoring in ci: A preliminary study on the perceived advantages and barriers. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 564–568. IEEE, 2018.
89. A. Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann Publishers Inc., 2005.
90. S. Zhang, D. Caragea, and X. Ou. An empirical study on using the national vulnerability database to predict software vulnerabilities. In *International Conference on Database and Expert Systems Applications*, pages 217–231, 2011.