The Secret Life of Software Vulnerabilities: A Large-Scale Empirical Study

Emanuele Iannone, Roberta Guadagni, Filomena Ferrucci, Andrea De Lucia, *Senior Member, IEEE*, and Fabio Palomba

Abstract—Software vulnerabilities are weaknesses in source code that can be potentially 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 software engineering life cycle, 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 at which these methods should aim. To investigate the life cycle of known software vulnerabilities, we focus on how, when, and under which circumstances the contributions to the *introduction* of vulnerabilities in software projects are made, as well as how long, and how they are *removed*. We consider 3,663 vulnerabilities with public patches from the National Vulnerability Database—pertaining to 1,096 open-source software projects on GitHub—and define an eight-step process involving both automated parts (e.g., using a procedure based on the SZZ algorithm to find the vulnerability-contributing commits) and manual analyses (e.g., how vulnerabilities were fixed). The investigated vulnerabilities can be classified in 144 categories, take on average at least 4 contributing commits before being introduced, and half of them remain unfixed for at least more than one year. Most of the *contributions* are done by developers with high workload, often when doing maintenance activities, and *removed* mostly with the addition of new source code aiming at implementing further checks on inputs. We conclude by distilling practical implications on how vulnerability detectors should work to assist developers in timely identifying these issues.

Index Terms—Software vulnerabilities, mining software repositories, empirical software engineering

1 Introduction

Software vulnerabilities are flaws in the design, implementation, or operation management of a software system that can be exploited to break through security policies [1], possibly causing loss or harm [2], [3], [4]. The risks associated with vulnerabilities on software systems are extreme: for instance, in 2017, the WannaCry Ransomware Attack [5] 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 businesses and users over six trillion USD [6] and will affect an even larger variety of software systems, ranging from public physical tests [7] to data-intensive applications [8], and more. For these reasons, keeping software security under control is still one of the main concerns of developers and organizations [9], [10].

To address this concern, software organizations and teams are adopting the McGraw's [11] advice to *"build security in"* rather than waiting until vulnerabilities are discovered in running software [12]. At the same time, researchers have been proposing novel methods and tools to facilitate the identification of software vulnerabilities [13], [14], [15], [16]. Nevertheless, recent empirical studies [17], [18], [19], [20], [21] provided evidence that developers are concerned with vulnerabilities and their potential effects, but such vulnerabilities still often occur in practice.

A possible reason behind the diffuseness of vulnerabilities may be the lack of empirical knowledge of their life cycle, i.e., when developers introduce vulnerabilities, during which development activities vulnerabilities are more prone to be introduced, or how developers remove vulnerabilities in practice. An improved understanding of these aspects may inform the software engineering community and tool vendors on how to better support developers in both the vulnerability identification and fixing process [22], for instance, by devising novel mechanisms or adapting currently available detection approaches to identify vulnerabilities when they are typically introduced. Moreover, the analysis of the life cycle of vulnerabilities can uncover patterns that would be helpful for the deployment of best practices [23] that help improving security processes [24].

To bridge this knowledge gap, this article presents a large-scale empirical study on the life cycle of software vulnerabilities. We mine the National Vulnerability Database (NVD) [25] (the U.S. government repository of standard vulnerability management data) to extract a set of vulnerabilities for which their fixing commits are public. Overall, we study 3,663 vulnerabilities of 144 categories across 1,096 different GitHub projects. We first automatically identify the corresponding *vulnerability-contributing commits* (VCCs)—i.e., the set of commits that

• *The authors are with the Software Engineering (SeSa) Lab, University of Salerno, 84084, Fisciano, Italy. E-mail: r.guadagni1@studenti.unisa.it, {eiannone, fferrucci, adelucia, fpalomba}@unisa.it.*

Manuscript received 12 April 2021; revised 28 December 2021; accepted 3 January 2022. Date of publication 6 January 2022; date of current version 9 January 2023.
The work of Fabio Palomba was supported in part by Swiss National Science Foundation - SNF Project under Grant PZ00P2_186090 (TED).
(Corresponding author: Emanuele Iannone.)
Recommended for acceptance by M. Nagappan.
Digital Object Identifier no. 10.1109/TSE.2022.3140868

contribute to the introduction of a vulnerability [26]— to investigate (i) how vulnerabilities are contributed to, (ii) by whom, and (iii) under which circumstances. Then, we conduct a survival analysis to evaluate how long they remain in the considered software systems and verify, adopting an open coding methodology, how the fixing process takes place.

The key results of our empirical study show that developers mostly contribute to vulnerabilities while doing maintenance activities, such as fixing bugs. Furthermore, vulnerabilities remain in a project's codebase for long time—half of them for at least 511 days passing through 9 changes—and are removed by developers with simple code changes, such as escaping HTML entities. Our findings provide evidence that developers, even the most expert ones, do need help to detect vulnerabilities earlier, for example, through better tool support. In addition, vulnerability detectors should be context-dependent and take into account what a developer is doing when suggesting potentially vulnerable code. To sum up, the paper provides the following main contributions:

1) A large-scale empirical investigation on how and under which circumstances vulnerabilities are contributed to, how long they survive, and how they are removed on thousands of projects written in different programming languages and concerning different application domains;
2) A large curated dataset on software vulnerabilities [27], along with the whole set of scripts used to perform our analyses, that can be exploited by other researchers to build upon our study and further explore the problem of software vulnerabilities;
3) A research roadmap describing the next steps and challenges that the software engineering research community should face to provide better support to practitioners.

Structure of the Paper. Section 2 describes the research questions driving our empirical study and the methodology to address them. Section 3 presents the results, while Section 4 overviews the main implications and lessons learned. Section 5 discusses the limitations of our paper and how we mitigated them. Section 6 reports the related literature. Finally, Section 7 concludes the paper.

2 Research Methodology

This section describes the methodology employed to address the research goals driving our investigation.

2.1 Research Goals and Questions

The *goal* of the study was to investigate the life cycle of software vulnerabilities, with the *purpose* of assessing when and how they are introduced and fixed by developers. The *perspective* is of researchers and practitioners: the former are interested in better understanding the phenomenon of software vulnerabilities and explore the characteristics behind vulnerability-contributing and fixing commits; the latter are interested in acquiring information that may be useful to inform and improve their security process.

Our study was structured around four main research questions (**RQ**s). We started analyzing *how* vulnerabilities are contributed to, particularly to understand whether the contribution takes place with the creation of a new file or during maintenance and evolution activities performed on existing files. Such an analysis provide a general overview on the nature of vulnerability-contributing commits, which could be valuable for vulnerability detectors designers to comprehend the properties that characterize VCCs; for example, should vulnerabilities be mainly contributed when the affected file is created, an approach pinpointing potential vulnerabilities when a new file is introduced in the repository would be worthwhile [28], [29]. This reasoning led to our $\mathbf{RQ}_1$:

> **$\mathbf{RQ}_1$.** *How contributions to vulnerabilities are made into the source code?*

Second, we studied under which circumstances vulnerabilities are more likely to be introduced by developers. We focused on three aspects: (1) *commit goal*, i.e., the intended action that a developer was performing when contributing to vulnerabilities, e.g., the implementation of new features or a refactoring activity; (2) *project status*, i.e., the proximity of a VCC to new releases and to the project's startup; and (3) *developer status*, i.e., considering the authors' experiences and workloads when they contributed to the introduction of a vulnerability. These three aspects are key to describe the *context* and the *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 [30], [31], but also to provide developers with details that can help to understand the reasons leading to a vulnerability introduction [32], [33]. Hence, we asked:

> **$\mathbf{RQ}_2$.** *What is the context in which contributions to vulnerabilities are made into the source code?*

After describing the circumstances leading to the introduction of vulnerabilities, we moved toward the understanding of their *longevity* by means of a survival analysis method [34]. This helped to describe the lifespan of vulnerabilities: on the one hand, this is useful to understand for how long such vulnerabilities can be exploited by attackers, thus challenging previous findings in the field [23]; 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 of unfixed vulnerabilities [35], [36]. Thus, we asked:

> **$\mathbf{RQ}_3$.** *What is the survivability of vulnerabilities?*

Finally, we investigated the vulnerability fixing process. Such an analysis can help researchers understanding how to better support developers with (semi-)automated techniques to remove vulnerabilities or to mitigate their effect [37], [38], [39], e.g., by defining a novel catalog of vulnerability-specific refactoring operations. We asked:

TABLE 1
Distribution of CVEs With Respect to the *2020 CWE Top 10 Most Dangerous Software Weaknesses*

Name	Description	Total
CWE-79 – Cross-site Scripting	The software does not properly neutralize user-controllable input before it is placed on a web page served to other users.	12.45%
CWE-787 – Out-of-bounds Write	The software writes data beyond the bounds of a buffer.	2.87%
CWE-20 – Improper Input Validation	The software does not properly check whether the input can be processed safely and correctly.	7.45%
CWE-125 – Out-of-bounds Read	The software reads data beyond the bounds of a buffer.	7.54%
CWE-119 – Improper Restriction of Operations within the Bounds of a Memory Buffer	The software perform operations (such as reads or writes) beyond the bounds of a buffer.	8.98%
CWE-89 – SQL Injection	The software does not properly neutralize special elements when building an SQL command using externally-influenced input.	2.84%
CWE-200 – Exposure of Sensitive Information to an Unauthorized Actor	The software exposes sensitive information to an actor that is not explicitly authorized to have access to that information.	4.83%
CWE-416 – Use After Free	The software references memory after it has been freed.	2.79%
CWE-352 – Cross-Site Request Forgery	The web application does not properly verify whether a valid request was intentionally provided by the requesting user.	1.56%
CWE-78 – OS Command Injection	The software does not properly neutralize special elements when building an OS command using externally-influenced input.	0.96%
		52.27%

RQ$_4$. *How are known vulnerabilities removed from the source code?*

The following sections describe the context of the investigation and how we addressed those research questions.

2.2 Context Selection

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

Vulnerabilities. We analyzed vulnerabilities from the National Vulnerability Database (NVD) [25], which was created by the U.S. NIST Computer Security Division [40] to collect and provide public information about known vulnerabilities affecting software systems and their causes. We relied on NVD for three main reasons:

1) it includes a comprehensive set of publicly known vulnerabilities, each described through CVE (Common Vulnerabilities and Exposure) [41] records, enriched with additional information, such as external references, the severity (Common Vulnerability Scoring System - CVSS), the related weakness type (Common Weak Enumeration - CWE) and the known affected software configurations (Common Platform Enumerations - CPEs);
2) it aggregates data coming from multiple sources;
3) it has been exploited by various research communities in the past [42], [43], [44], [45].

We selected the vulnerabilities having both the fixing commit (i.e., the one that officially patched a publicly disclosed vulnerability) and the project's GitHub repository available—otherwise, we could not address our research questions, as explained later in this section. As a result, the initial set of 142,546 CVEs was reduced to 3,663 CVEs, belonging to 144 distinct weakness types (CWEs). Further details on how they were collected are described in Section 2.3. Table 1 gives an overview on the distribution of the CVEs grouped by their CWE appearing in the *2020 CWE Top 10 Most Dangerous Software Weaknesses* [46] (henceforth, *CWE Top 10*). This widely-known list describes the most common and impactful security issues affecting software systems. We relied on it to enrich the analyses linked to our **RQ**s and so providing additional insights on peculiar characteristics of specific types of vulnerabilities.

Systems Histories. The considered vulnerabilities pertain to 1,096 GitHub projects. These projects have an average of 10,214 commits, 240 contributors (i.e., developers who contributed with at least one commit) and 3.34 CVEs. The full list of projects, along with their main characteristics, is reported in our online appendix [27].

2.3 Data Extraction

Fig. 1 overviews our data extraction process, which we describe in the following.

Mining NVD. We exploited CVE-Search [47], an open-source tool that imports the entire set of CVE Records (a.k.a., CVEs) from the NVD repository into a MongoDB database for easier search and processing. We obtained the full JSON dump of the extracted records on the date of 1st November 2020, which comprised a total of 142,546 CVEs. We filtered out those records that did not report any GitHub link to a commit—in these cases we could not identify the affected project nor any vulnerability-contributing commit. We also discarded the CVEs that reported commits to different GitHub projects, since we could not know where the vulnerability was effectively residing. These two steps caused the removal of 137,470 (96.44%) and 74 (0.05%) CVEs, respectively. Moreover, we ensured that the fixes were not merging commits, which, by their nature, do not apply any modification in the project history and merely incorporate the changes (i.e., a set of commits) from a branch into another [48]. We could not consider them as actual patches since we are interested in the moments when fixes are added into the history rather than the moment in which they are sent into the main branch (Step 1 in Fig. 1). This filter removed an additional set of 814 (0.57%) CVEs whose referenced commits were all flagged as merging commits.

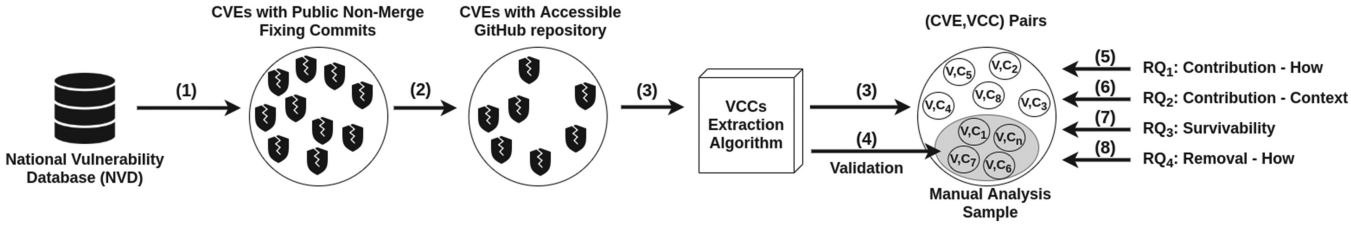


Fig. 1. Process for extracting the data needed to answer our research questions.

Mining GitHub*Repositories.* We mined the GitHub repositories of the projects in which the selected vulnerabilities appeared so that we could access their history information; in this case, we had to discard the vulnerabilities whose projects' repository were unavailable (e.g., the access was restricted to authorized users only), this caused the removal of 525 (0.37%) CVEs. For this reason, we ended up with 3,663 CVEs pertaining to 1,096 projects (see Step 2 in Fig. 1).

Identification of Vulnerability-Contributing Commits. For each vulnerability, we extracted the set of VCCs by following the principles of the SZZ algorithm [49], which fetches the commits that last changed the deleted lines in a given bug-fixing commit. Since traditional bug fixes follow different mechanisms from the ones adopted for software vulnerabilities due to their nature, we opted for defining our data extraction procedure to mine the VCCs to improve the performance of basic SZZ (Step 3 in Fig. 1). Specifically, for each file A in the set of files modified within a vulnerability-fixing commit f, the algorithm (1) obtains the `git-diff` with respect to the previous commit, (2) retrieves the list of modified—deleted or added—lines in A, (3) blames all the deleted lines in A using `git-blame` command to obtain the commits where those lines were changed last, (4) blames the contextual lines (up to three lines before and after) of continuous blocks of changes made only of added lines. On the one hand, blaming the deleted lines allows fetching those commits that are likely the ones that added the flawed lines—so, contributing to the vulnerability; on the other hand, continuous blocks of new lines may represent the addition of checks (e.g., `if` or `try-catch` constructs) whose absence enabled the vulnerability, and so the blames of the contextual lines retrieves the commits that added those code snippets that lacked proper controls mechanisms. When running the algorithm, we carefully avoided blaming empty and comment lines [50], as well as irrelevant non-source code files (e.g., documentation, build, blob, and test files), since their changes are unlikely to contribute to a vulnerability—before excluding the build files, we made sure not to have any build-related vulnerabilities in our dataset. In this respect, we adopted some filters based on the use of regular expressions onto the entire file path. For instance, we did not consider files having *'test'* suffix placed in a directory named `test`. In addition, we did not consider merging VCCs, as they do not report any actual modifications (thus, applying the same rationale used to discard invalid fixing commits from NVD). What is more, we avoided blaming the contextual lines of change blocks made only of new functions or methods, as they could be placed anywhere in the source code. Specifically, we obtained the list of functions and methods added within the commit by parsing the diff content via Lizard,[1] a library that offers parsing capabilities of source code files written in many different programming languages, such as C, PHP, Ruby, Scala, JavaScript. This heuristic was not applied to files whose language was not supported by Lizard. Whenever a vulnerability required multiple commits to be patched, the full set of VCCs was determined by the union of the results returned by our extraction procedure on each fixing commit. It is worth noting that `git-blame` is able to automatically follow the origin of the blamed lines across file renames—i.e., when the file changes its path, but its content does not. The limitations of this mechanism are described in Section 5. The identification of VCCs relied on the PyDriller repository mining library [51], which offers an implementation of standard SZZ algorithm, on which we added our heuristics. We implement the algorithm as a set of Python scripts, that we released in our online appendix [27]. Overall, the VCCs extraction procedure was able to identify a total of distinct 12,256 VCCs.

2.4 Data Validation

Most of the analyses needed to address our research questions depend on the precision of the VCCs identification mechanism. Recent work has criticized the capabilities of SZZ [52]; for this reason, we manually validated the effectiveness of our VCCs mining procedure on our dataset before proceeding (Step 4 in Fig. 1), as it is based on the same assumptions used by SZZ about the `git-blame` capabilities to fetch the contributing commits. Since the VCCs extraction procedure mapped a set of VCCs to each CVE in our input dataset, the complete output of the algorithm can be described as a list of pairs (v, c), where v represents a CVE, and c one of the VCCs extracted from v. This list represents the population to be evaluated, made of a total of 17,239 pairs. Two of the authors manually labeled the validity of the pairs, i.e., given a pair (v, c), the inspectors had to understand whether the changes applied by the VCC v actually contributed to the introduction of the vulnerability associated with the CVE c. In other words, a positive classification means that, according to the inspectors' views, the algorithm was able to mine a real contributing commit of that vulnerability. To this end, the two inspectors were given access to both the GitHub link to the VCC, and the NVD link to the CVE description—in which the fixing commit(s) could be found as well. In this respect, it is worth remarking that the inspectors did not limit their analyses to the given VCCs, but also navigated back from them intending to verify that these commits were not false positives—

1. https://pypi.org/project/lizard/

this is an operation recommended by previous works that validated the performance of SZZ [50]. The entire process was conducted on a statistically significant sample of 376 pairs (confidence level = 95%, margin of error = 5%). Before starting, the two authors held an initial round of alignment on a separate set of 126 pairs (equal to $1/3$ of the sample size), so that the two could similarly label the pairs. After this phase, the two inspectors compared their assessments, finding an agreement in 119 cases, representing the 94.44% of the total. To account for the possibility of agreements made by chance, we also computed Cohen's κ statistic [53], which measures the *inter-rater agreement* of the inspection task. We observed $\kappa = 0.817$ ($p = 0.05$), indicating a strong agreement [54], allowing us to equally distribute the sampled 376 pairs between the two inspectors.

The precision scored by our VCCs mining algorithm was 68% (namely, 254 correct mappings). The inspectors observed that most of the failures were not attributable to the algorithm itself but to how the fixing commits were made by developers. Indeed, they discovered many cases in which fixing commits were made of unrelated code changes [55], i.e., mixing vulnerability fixing changes with other activities, such as implementing new features or doing code refactoring. Such commits are known as *tangled changes*, characterized by a large number of touched files—ranging from dozens to hundreds. For example, the vulnerability described by CVE-2015-2861 and affecting project VESTA CONTROL PANEL, is patched by the commit `527e4a9a`,[2] which both solves a CSRF vulnerability—by implementing a token validation mechanism—and update the graphical user interface, as stated by the commit message *'UI update'*. Such tangled commits caused our VCCs mining procedure to blame a large set of contributing commits, likely unrelated with the actual patch applied, so hindering the precision of our approach.

In any case, our validation and the above considerations report that the quality of the exploited dataset is still good enough to allow a valid reporting of the life cycle of software vulnerabilities.

2.5 RQ_1. How - Research Methodology

To address $\mathbf{RQ_1}$, we first computed the number of VCCs required to introduce each vulnerability, presenting the results via descriptive statistics (Step 5 in Fig. 1). We also investigated the typical duration of the *"insertion window"*—i.e., the time ranging from the first and last VCCs (henceforth, the *turning point*)—for those vulnerabilities requiring more than one commit to be introduced in the source code. In this way, we could provide a high-level overview on the basic characteristics of VCCs.

Then, for each file affected by a vulnerability, we computed the number of commits between their creation and the first VCC, as well as between the creation and the turning point. This lets us verify whether certain files contributed to the activation of a vulnerability since their introduction or only as a side effect of multiple changes during maintenance and evolution. In addition, we analyzed the common size of VCCs, both in terms of touched files and code churn.

We also made these two-perspective analyses from the point of view of the *CWE Top 10* in order to identify the existence of particular vulnerability types that exhibit trends diverging from the general one. To provide high-focused discussions, we only report and discuss the results coming from the *CWE Top 10*. Our online appendix [27] reports the raw results for the entire *CWE Top 25*.

2.6 RQ_2. Context - Research Methodology

To study the *context* in which developers contribute to the introduction of vulnerabilities, we automatically classified each VCC to one or more of the categories described by Tufano *et al.* [22] (see Table 2). These categories are divided into three groups: (1) the *commit goal*, i.e., the task the developer was performing when contributing to a vulnerability; (2) the *project status*, i.e., the development phase of the project when the contribution to a vulnerability was made with respect to releases and the repository creation date; (3) the *developer status*, i.e., the characteristics of the developer who contributed to a vulnerability. The classification of the VCCs allowed us to analyze the typical circumstances leading to the introduction of vulnerabilities. When performing this analysis, we had to discard 403 VCCs (hence considering 11,853 out of the total 12,256) for two reasons. On the one hand, not all repositories make use of releases (i.e., `git-tag` mechanism) and, therefore, we could not assign the *project status* tags to commits belonging to those projects. On the other hand, we could not consider the cases where a contributing commit was amended or rebased since these operations change the original commit date, leading to negative distances in days. In our online appendix [27] we explicitly marked those commits for the sake of replicability. We point out that the removal of these 403 VCCs led to the removal of 607 vulnerabilities (as the set of their contributing commits overlap). In addition, we provided a closer inspection on the *CWE Top 5* due to space limitations. The remainder of the CWEs is reported in our online appendix [27]. The automatic classification approach works as follows (see Step 6 of Fig. 1):

Commit Goal. We relied on a previously proposed approach that analyzes commit messages to check for the presence of keywords indicating specific development activities [56], [57]. For instance, a commit was classified as *bug fixing* if the corresponding message contained keywords like *'bug'*, *'defect'*, *'fix'*, etc. With this approach, it is possible to have commits belonging to multiple goals—the complete list of keywords used is available in our appendix [27].

*Project Status.*For each VCC, we computed the number of days separating it from the date of the nearest minor or major release and assigned the proper *working on release* category. We followed a similar procedure for *project startup* category, computing the number of days between the first commit onto the repository and the VCC [57].

Developer status. We considered two different perspectives. First, we computed the *workload* [22] of the developers who made the VCCs. To compute such value, we chose two different proxy metrics: (1) number of commits, representing the number of different contributions made by developers, and (2) code churn, representing the size of the code changes. In other words, the higher the number of

2. https://github.com/serghey-rodin/vesta/commit/527e4a9a62204be9b34c1338fadfe959b0fd3974

TABLE 2
Tags Assigned to VCCs to Answer RQ_2

Tag	Description	Values
	Commit Goal	
New feature	The VCC aimed at implementing a new feature in the system.	[true, false]
Bug Fixing	The VCC aimed at fixing a bug.	[true, false]
Enhancement	The VCC aimed at enhancing the system.	[true, false]
Refactoring	The VCC aimed at performing refactoring operations.	[true, false]
	Project Status	
Working on Release	The VCC was performed [value] of issuing a minor or major release.	[the same day, the day before, within the week, within the month, over one month before]
Project Startup	The VCC was performed [value] the starting of the project.	[the same week of, the week after, the month after, over one year after]
	Developer Status	
Commit Workload	The developer had a [value] workload, in terms of commits, when the VCC was performed.	[low, medium, high]
Churn Workload	The developer had a [value] workload, in terms of code churn, when the VCC was performed.	[low, medium, high]
Tenure	The developer was a [value] when the VCC was performed.	[newcomer, medium, expert]

commits—or code churn analogously—the higher the workload the developer had on a given time window. Specifically, given a VCC performed by a developer c at date d, we computed the workloads distributions—both in terms of commits and code churn—for all of the developers of the project within the 30 days before the date d. Then, we apply a min-max scaling [58] on the two distributions to scale their ranges in [0,1]. At this point, the *commit workload* of c was assigned *'low'* if her number of commits was strictly lower than 0.25, *'medium'* if between 0.25 and 0.75, and *'high'* when higher than 0.75. An analogous mapping was applied for the *code churn workload*.

Second, we computed the *tenure* of the developers with respect to the project [59], [60], which is an experience metric that counts the number of months from the developer's first contribution to the project to the moment in which she made a given VCC. Similarly to the two *workload* metrics, we scale the distribution of all developers' tenures, and remapped the values *'newcomer'*, *'medium'*, *'expert'* by using the same splitting criterion. We also correlated the experience of developers to the CVSS score [61]—i.e., the de-facto standard to measure the overall severity of a vulnerability—of the CVEs to which they had contributed, by exploiting the Spearman's rank correlation test [62]. Such a test can detect the presence of correlations between two ordinal variables without making any assumption on their distribution.

2.7 RQ_3. Survivability - Research Methodology

By the use of contributing and fixing commits, we could analyze the survivability of each vulnerability. In the case of vulnerabilities with more than one VCC, we considered the turning point commit (i.e., the last VCC) for this analysis, as it marks the moment in which a vulnerability starts living; while in the case of multiple fixing commits, we considered the last one, as it is the moment in which a vulnerability ceased to exist. Given a vulnerability, we defined the time interval between its turning point and its last fixing commit as the *vulnerable period*. We performed a survival analysis [34], employing a statistical method modeling the time duration of a subject until one or more events of interest happen. The survival function $S(t) = P\pi(T > t)$ indicates the probability that a vulnerability (i.e., our subject) 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 $t \rightarrow \infty$, $S(t) \rightarrow 0$. 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 subject population consists of vulnerability instances while the event of interest is represented by their last fixing commit (i.e., their 'death'). Thus, the *lifetime* (or *time-to-death*) of a vulnerability is the duration of the vulnerable period, measured by both (1) the number of elapsed days and (2) the number of commits touching the files involved in the fixing commits (Step 7 in Fig. 1). We considered only the changes applied onto the files responsible for the vulnerability as we were interested in the number of opportunities the developers had to identify and fix the security flaw but failed. If we had considered the total number of commits we would risk making wrong conclusions for repositories with a high-frequency contribution rate, such as Linux. Ultimately, both these metrics complement each other: for instance, two changes may occur in a short interval of days, but with a high number of changes in-between, so analyzing only one of the two would have been limiting.

We investigated survivability from two different perspectives. On the one hand, we analyzed the general survivability of all of the vulnerabilities against the survival distributions provided by the vulnerabilities grouped by the *CWE Top 10* (the comparison with the entire *CWE Top 25* is available in our online appendix [27]). On the other hand, we compared the general survivability with the trends of the vulnerabilities grouped by the main programming language of the affected repository. To this end, we relied on the GitHub API for detecting the predominant language, so there could be cases in which the projects have no specific

language (i.e., N/A), which we still included in this analysis. The analyses were conducted exploiting the Kaplan-Meier estimator [63], a non-parametric survival analysis method, suitable to our survival periods as we could not assume any particular distribution. We relied on the implementation provided by LIFELINES[3] package for PYTHON.

2.8 RQ_4. Removal - Research Methodology

To address $\mathbf{RQ}_4$ (Step 8 in Fig. 1), we conducted an open coding process [64] over the pairs (f, v), where f represents one of the fixing commits that patched the vulnerability described by the CVE v. This process consisted in assigning a label (i.e., the *code*) to each pair to summarize the actions the developer undertook to resolve the vulnerability. A complete inspection of all the population of 3,983 pairs would have been impractical, so we decided to carry out the coding process on a statistically significant sample made of 351 pairs (confidence level = 95%, margin of error = 5%). These pairs were equally distributed between two authors of the paper who independently and manually categorized the kinds of actions performed by developers (e.g., the addition of a precondition check) by relying (i) on the commit message, (ii) on the CVE description, and (iii) on the diff content. Afterward, 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 outcome of this process consisted of the definition of a taxonomy of methods explaining how known vulnerabilities are removed from the code. Finally, we zoomed into each fixing categories, by reporting the descriptive statistics for the main characteristics of the sample fixing commits, such as the average number of touched files. We also connected this taxonomy to both the *CWE Top 10* and the projects' language, to find relationships describing strong connections between fixing methods and weakness types/languages.

3 ANALYSIS OF THE RESULTS

This section presents the results of our research questions.

3.1 RQ_1: How Contributions to Vulnerabilities are Made Into the Source Code?

The Vulnerabilities' Perspective. Among all 3,663 vulnerabilities, our VCCs extraction procedure found a total of 12,256 VCCs among the 1,096 projects. Analyzing the number of contributing commits needed to fully introduce a vulnerability, we observed that the average number of VCCs per vulnerability was 4.71 (median = 2). In particular, 2,232 out of 3,663 vulnerabilities (corresponding to the 60.93%) required more than VCCs to be introduced. This first result lets us see that, in the majority of the cases, software vulnerabilities are not added within a single commit, but may require a variable number of evolutionary activities. Zooming into these 2,232 vulnerabilities, we discovered that the size of the "*insertion window*" ranges from 0 to 8,566 days (i.e., over 20 years), with an average of 1,520 days (about 4 years). This means that the first contribution to a vulnerability may happen way before the final contribution that enabled it. In summary, vulnerabilities are not generally introduced because of a single mistake of a developer, but rather, due to an accumulation of errors that introduced weak code elements—which, together, made up the vulnerability in the form with which it was discovered and fixed.

There is no shortage of extreme outliers: the vulnerability with the largest number of VCCs is an SQL Injection (CWE-89) in the project EXPONENT CMS (CVE-2016-8897),[4] reaching the value of 205. By inspecting its only fixing commit, we found that the amount of both added and deleted lines was very high (over 26,800,000 and 52,000, respectively), spanning across over 40 different files. Indeed, its commit message states *'iniitial effort to greatly enhance system security (xss, sql inject, file exploit, rce, etc...)'*, clearly indicating its goal to reimplement the security checks in many different files. This may explain why our algorithm was able to discover over 200 contributing commits. Going more in-depth, the CVE associated with this vulnerability mainly refers to the file `helpController.php`, but actually, the same type of defect was found to be spread across multiple source files (e.g., `expSettings.php`, `donation.php`), which did not properly manage the user-supplied data for the execution of SQL queries. What is more, this fix patches other flaws related to improper input validations, such as XSS vulnerabilities, outside the scope of CVE-2016-8897. By inspecting its VCCs, we can see that they consisted of additions of new SQL queries not carefully protected against possible injections. This large number of weak queries suggests a lack of awareness by the contributors of EXPONENTCMS about the problem of SQL injection.

Table 3 shows on its left side the main descriptive statistics on the number of contributing commits needed to introduce each vulnerability. The table also focuses on the vulnerabilities belonging to the *CWE Top 10*. In this respect, CWE-89 ('SQL Injection') is confirmed to be the vulnerability type that generally requires a large number of VCCs to be fully introduced. This result is somehow unexpected. Although SQL injection is one of the most studied security issues [65], [66], it is often neglected by developers [67], who are very likely to add many instances of SQL injections across different evolutionary activities, as shown by CVE-2016-8897. Nonetheless, as soon as the problem is discovered, patching the weak queries happens to be an effort that can be done within the context of a single, despite large, change. The whole story is quite similar for CWE-352 ('Cross-Site Request Forgery'), likely because CSRF vulnerabilities are caused by the improper or the lack of checks to establish whether HTTP requests—related to sensitive actions such as a money transfer transaction—are sent intentionally or not. Hence, it is quite reasonable that fixes of CSRF may touch many files, and so increasing the number of discovered VCCs.

The Files' Perspective. We identified a total of 7,318 files changed within the 12,256 VCCs. Most of the times, vulnerabilities are (partially) introduced in commits where developers apply changes on a few files—indeed, on average, the number of files modified per VCC is 1.43 (median = 1), while in 62 cases (0.51%) the VCCs modified more than 10 files. This is somehow

3. https://pypi.org/project/lifelines/

4. https://nvd.nist.gov/vuln/detail/CVE-2016-8897

TABLE 3
Summary of Descriptive Statistics of Both the Number of VCCs ($N = 12,256$) of Each Vulnerability (Left), and the Number of Files Touched by Each VCC (Right)

	CVEs	VCCs per CVE				VCCs	Files per VCC			
		Mean	Min	Med	Max		Mean	Min	Med	Max
Overall	3,663	4.706	1	2.0	205	12,256	1.426	1	1.0	819
CWE-79	456	5.094	1	2.0	134	1,767	1.452	1	1.0	26
CWE-787	105	5.114	1	2.0	200	489	1.337	1	1.0	34
CWE-20	273	4.103	1	2.0	60	936	1.439	1	1.0	33
CWE-125	276	3.362	1	2.0	20	675	1.153	1	1.0	11
CWE-119	329	3.146	1	2.0	22	823	1.140	1	1.0	19
CWE-89	104	18.356	1	4.0	205	991	3.054	1	1.0	819
CWE-200	177	3.701	1	2.0	59	584	1.233	1	1.0	16
CWE-416	102	2.677	1	1.5	19	255	1.177	1	1.0	22
CWE-352	57	9.088	1	3.0	100	498	1.456	1	1.0	25
CWE-78	35	5.257	1	3.0	50	181	1.088	1	1.0	4

All of the vulnerabilities are also grouped by the CWEs belonging to the Top 10 Most Dangerous Software Weaknesses.

in contrast with the findings of Hindle *et al.* [68], who reported that commits involving more than 10 files are more prone to introduce defects. Conversely, our findings corroborate the hypothesis that vulnerabilities are different from other types of software defects, as also pointed out by Morrison *et al.* [24]. In particular, VCCs tend to add new lines rather than delete existing ones. Indeed, the median numbers of added and deleted lines were 110 and 26, respectively. This result indicates that the contributions to a vulnerability seem to be caused by the insertion of new code rather than the removal of existing code, which is reasonable considering the nature of implementation-level vulnerabilities—i.e., the ones considered in this study—occur when new and flawed code is added.

Table 3 shows on its right side the main descriptive statistics on the number of files touched by each vulnerability-contributing commit. Similar to what we have seen with the vulnerabilities' perspective, the VCCs of type CWE-89 are further confirmed to be the ones with the largest number of outliers. For example, the commit `6e1f4e2b` of GeniXCMS,[5] which contributed to CVE-2016-10096, touched 819 different files (2,056 if we consider the ones we discarded from the study). Interestingly enough, despite its large size, this commit only contributed to the introduction of CVE-2016-10096, without affecting any other vulnerability. By analyzing the `git-diff` we can see that most of the new files are actually open-source libraries included within the source code. This is confirmed by the commit message, stating *'Update Summernote, Jquery-UI, Bootstrap, Vendor Library via Composer. Add new Vendor'*. However, the commit was not restricted to updating libraries, but also to other different types of changes, explaining why only a single vulnerability was introduced.

If we take into account the moment in which the files were added the first time in the project, we see that on a median the number of previous changes a file before being involved in the first VCCs is 0, raising to 29 if we consider the previous changes before the turning point commit. Indeed, 3,506 out 7,318 files (47.91%) turned out to be created within a VCC, of which in 456 of the cases (13.01%) that VCC is the only contribution that fully introduced the related vulnerability. All of this has to important implications: (i) half of the involved files are introduced already flawed, and (ii) in a non-negligible number of cases the vulnerabilities are caused by the addition of a flawed file. On the other side of the spectrum, we found files that underwent substantial modifications before they became part of the vulnerability. Specifically, a set of 58 out of 7,318 files (0.79%) were changed in more than 500 commits before they started becoming vulnerable. This happened, for example, in Total.js framework where the file `index.js`—a very large file containing the core login of the entire framework—was involved in a VCC of CVE-2019-8903 after 1947 commits since its creation. The issue was caused by an improper definition of a regular expression contained in `REG_TRAVEL` global variable intended to prevent path traversal attacks.

Main findings for RQ_1

On average, 4.71 VCCs were made to fully introduce a vulnerability. In most cases (60.93%), more than one VCC was made, spanning over about 4 years on average. SQL Injection was the vulnerability type that required the largest number of VCCs (18.36). Almost half of the files involved in the VCCs considered (47.91%) were created in the context of those commits, supported by the fact that, on a median, a VCC adds 110 lines and deletes 26 existing ones. However, VCCs seem not to touch many files (1.43 on average).

3.2 RQ_2: What is the Context in Which Contributions to Vulnerabilities are Made Into the Source Code?

Table 4 reports the frequency distribution of all 11,853 VCCs with respect to the categories defined by Tufano *et al.* [22], grouped by the *CWE Top 5* (the full results are available in our online appendix [27]). For this research question, we had to discard 607 CVEs—as explained in Section 2.6—resulting in considering 3,056 vulnerabilities.

Commit Goal. Out of the considered 3,056 vulnerabilities, 2,648 of which received at least one commit goal tag among the respective VCCs; as a consequence, the remaining 408 did not receive any commit goal. The missing categorization was either due to (i) empty commit messages (e.g., in the case of the commit `3ed852ee` contributing to CVE-2019-15141) or

5. https://github.com/semplon/GeniXCMS/commit/6e1f4e2b

TABLE 4
Percentage of VCCs ($N = 11,853$) Having the Tags Produced for Answering $\mathbf{RQ}_2$

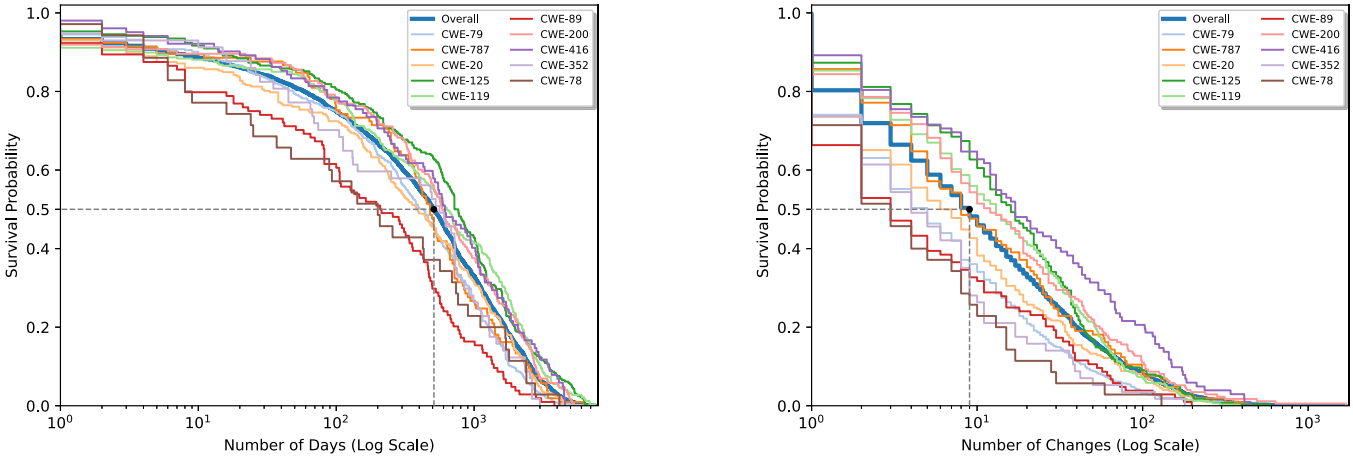
Tag Value		Overall	CWE-79	CWE-787	CWE-20	CWE-125	CWE-119
	New Feature	30.50%	27.62%	18.97%	29.78%	28.57%	32.23%
	Bug Fixing	37.06%	30.24%	52.71%	43.04%	32.94%	35.12%
	Enhancement	28.14%	22.01%	26.85%	31.34%	23.70%	27.69%
Commit Goal	Refactoring	19.23%	15.65%	14.29%	18.08%	17.48%	19.28%
	Same day	7.96%	14.90%	3.45%	10.79%	3.36%	4.68%
	One day before	2.47%	3.30%	0.99%	2.47%	2.18%	1.24%
	Within week	10.36%	10.72%	6.40%	13.00%	6.39%	7.71%
	Within month	19.53%	15.59%	21.92%	19.51%	13.45%	19.83%
Working on Release	Over 1M before	59.68%	55.49%	67.24%	54.23%	74.62%	66.53%
	Same week	3.17%	2.99%	1.97%	6.89%	1.85%	3.99%
	Week after	1.19%	1.75%	0.25%	2.99%	1.18%	0.55%
	Month after	11.65%	10.41%	4.68%	18.86%	9.41%	6.34%
Project Startup	Over 1Y after	83.99%	84.85%	93.1%	71.26%	87.56%	89.12%
	High	59.11%	67.21%	50.99%	58.52%	66.05%	49.59%
	Medium	10.79%	13.59%	2.96%	10.27%	9.92%	13.22%
Commit Workload	Low	30.10%	19.20%	46.06%	31.21%	24.03%	37.19%
	High	57.13%	65.46%	29.56%	58.00%	62.02%	49.04%
	Medium	7.41%	9.35%	7.39%	5.85%	10.25%	7.99%
Churn Workload	Low	35.46%	25.19%	63.05%	36.15%	27.73%	42.98%
	Expert	40.04%	42.39%	36.21%	39.40%	40.17%	35.26%
	Medium	16.72%	18.39%	19.21%	19.77%	19.83%	18.46%
Tenure	Newcomer	43.24%	39.21%	44.58%	40.83%	40.00%	46.28%

All of the VCCs are also grouped by the CWEs belonging to the Top 5 Most Dangerous Software Weaknesses.

(ii) no matching keywords (e.g., in the case of the commit `c2348ef6` contributing to CVE-2013-2091 where the message was *'Uniformize field country_id country_code country'*). At least half of the vulnerabilities match with 2 different categories, but we also experienced 569 vulnerabilities that matched all four categories, such as CVE-2017-17897, where two of its six VCCs state: (1) *'New: Super clean of permissions checks'*, and (2) *'Fix option STOCK_SUPPORTS_SERVICES'*. As reported in Table 4, when considering all vulnerabilities we see a quasi-uniform distribution among the four commit goals (with a slightly greater value for the *bug fixing* goal), indicating that a commit may contribute to a vulnerability during any development or maintenance activity. Specifically, the three maintenance goals (i.e., *bug fixing*, *enhancement* and *refactoring*) together form the great majority of the total (69.50%), in line with previous findings achieved when understanding how maintainability issues are introduced in open-source projects [22], [57], especially when they impact on several code entities and/or lines of code. More surprisingly, *refactoring* operations constitute the 19% of the total of VCCs, meaning that developers may fall into errors and introduce security-related problems while trying to improve the source code quality. While this may indicate that the side effect of refactoring might also include security issues [69] (besides maintainability problems [70]), we could not determine any cause-effect relations as the opposite ratio shows that out of all the refactoring commits mined, only the 0.07% were vulnerability-contributing commits. This is something that would deserve further investigation. From the point of view of the CWEs, the commit goal distributions follow the general trend, with only a notable difference for CWE-787 ('Out-of-bounds Write'), where more than half VCCs were done within the context of a bug fixing activity.

Project Status. Besides the commit goal tags, we discovered that most of the contributions to vulnerabilities happen quite distant from a release, with 59.68% of the cases issued more than 30 days before. This might indicate that their introduction is not strictly connected to the stress that developers may feel close to deadlines—as opposed to what happens with code smells [22]. Additionally, the vast majority of the VCCs (83.99%) are made after more than one year since the project startup, while only 11.65% within the first month of a new project. This result implies that (part of) vulnerabilities tend not to appear in newly-born projects; at the same time, they become more common when the project grows, possibly because of software aging [71]—this also explains why vulnerabilities typically affect projects after some time from their startup. Here too, CWE-787 diverges from the general trend, where even more vulnerabilities start appearing (i) after the first year of the project, and (ii) at least one month before a release. In addition, CWE-125 ('Out-of-bounds Read') increases these gaps even more. Both of the weaknesses types are related to memory management issues, predominantly linked to C/C++ programming language.

Developer Status. Next, we found that the majority of the cases VCCs' developers had a high workload when performing VCCs—both in terms of commits (59.11%) and code churn (57.13%)—hence corroborating the idea that the errors are commonly made by developers having a higher workload [22]. On the other hand, there is an almost equal distribution of both newcomer (43.24%) and expert developers (40.04%). While the presence of the formers could be owed to



(a) Survival analyses in terms of days.

(b) Survival analyses in terms of changes.

Fig. 2. Survival analysis of 3,645 vulnerabilities in terms of days (left) and changes (right), reported on a log scale. A subset of these vulnerabilities were also grouped by *CWE Top 10*.

a lack or poor awareness of the secure coding practices and the established projects' security policy [72], [73], the latter seems to confirm findings reported in the literature on the source code drawbacks introduced by expert developers [22], [74]. This may be explained by the fact that more experienced developers tend to perform more complex and critical tasks [75], strongly increasing the risk of introducing defects of any kind, including vulnerabilities [76]. We shed light on this regard by correlating the developers' tenure with the severity of the vulnerabilities they had contributed to, represented by their CVSS score. The Spearman's rank coefficient test reports a non-esistent correlation ($\rho = 0.002$) with no statistical significance at all ($p = 0.893$). This does not allow us to conclude anything about the relations between the developers' experience and the severity of the vulnerabilities to which they contributed, and further analyses using different metrics are needed.

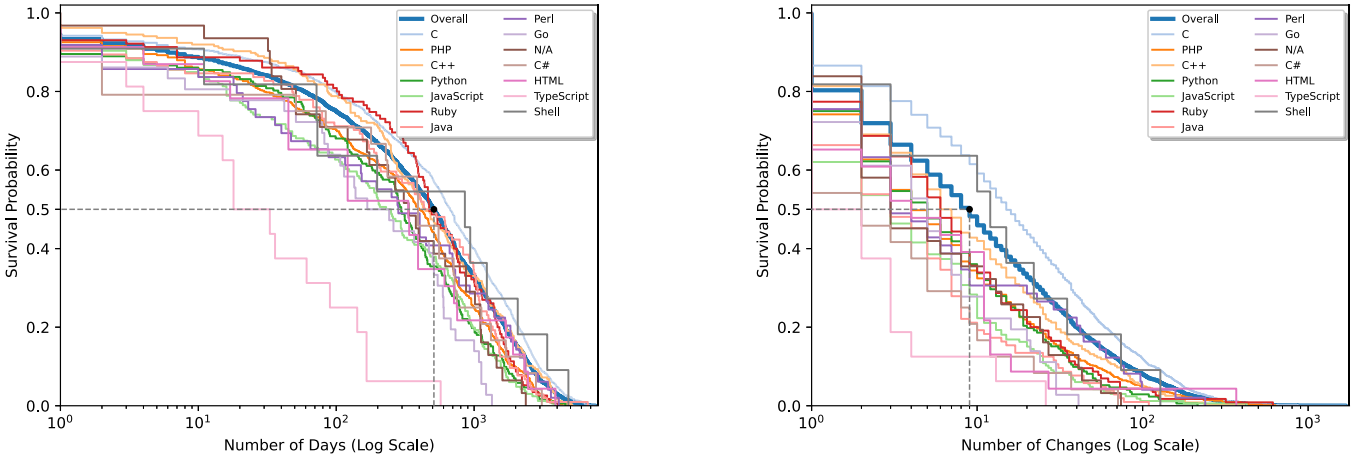
Main findings for RQ$_2$

The great majority of the VCCs (69.50%) included at least one maintenance activity (i.e., *bug fixing*, *enhancement* or *refactoring*) in their goal. The vast majority of vulnerabilities (83.99%) start appearing after the first year of the project's creation, generally (59.68%) issued at least 30 days before releases. There is an equal distribution of newcomer and expert developers (43.24% and 40.04%, respectively); however, over 55% developers performed VCCs when they had high workload, in terms of commits and code churn. This trend is not followed by vulnerabilities of type CWE-787, where over half of VCCs included bug fixing activities.

3.3 RQ$_3$: What is the Survivability of Vulnerabilities?

Considering all 3,645 vulnerabilities, we observe that a median of 9 changes are required to fix a vulnerability, distributed along 511 days (i.e., about one year and a half). The story is different if we consider the means. On the one hand, the needed days increase to 947.66 (i.e., over two years and a half); on the other hand, the number of changes reaches 31.77. This phenomenon is due to the presence of extreme outliers, i.e., vulnerabilities whose exposition period lasts far longer than the others. For instance, the vulnerability described by CVE-2014-3158, caused by an improper restriction of operations within the bounds of a memory buffer (CWE-119) in the PPP project, took 7,568 days (i.e., over 20 years) to be fixed. On the other side of the spectrum, CVE-2018-12684, caused by an exposure of sensitive information (CWE-200) in the CIVETWEB project, required 1,692 changes to file `civetweb.c` before its removal. It is worth noting that the fixing commit consisted of small program changes, namely a change of two relational expressions. Thus, it seems that all those changes were not instrumental for the patch but were done as part of other kinds of changes, corroborating the fact that the developers were not aware of the presence of the vulnerability. As a matter of fact, the vulnerable file—by the time of the fixing commit—was made of over 19,000 lines, so probably containing most of the logic of the entire application, explaining such a large number of modifications. From the above results, we can delineate two observations. First, vulnerabilities typically take years to be fixed and are, therefore, exposed to possible exploitations for a long time. Second, and perhaps more interestingly, there are cases in which the developers are neither aware of the presence of vulnerabilities nor have the proper instruments to identify and refactor them.

Fig. 2 depicts both the general survivability of all vulnerabilities and the individual survivabilities of their CVEs grouped by the *CWE Top 10* (the survival plots for the *CWE Top 25* is available in our online appendix [27]). Looking at the number of days (Fig. 2a), we see that almost all curves follow the same trend dictated by the *'Overall'* one—in which at 511 days there is an equal probability that the vulnerability is going to be fully fixed or not. We also observe that weakness types related to input validation issues (i.e., CWE-79, CWE-20, CWE-89, and CWE-78) exhibit a trend with lower values, implying the need for fewer days to fix a vulnerability. This result was somehow expected, as input validation flaws are far more common in web applications [77], encouraging researchers and practitioners to invest efforts on this issue and develop many automated testing techniques to detect them [78], [79], [80]. Indeed, the current state-of-the-practice



(a) Survival analyses in terms of days.

(b) Survival analyses in terms of changes.

Fig. 3. Survival analysis of 3,645 vulnerabilities in terms of days (left) and changes (right), reported on a log scale. A subset of these vulnerabilities were also grouped by the *Programming Language* having at least 10 CVEs.

in static application security testing (SAST) provides many automated solutions to detect issues related to the improper validation of externally-supplied inputs [81]—such as CppCheck [82], Flawfinder [83], or SpotBugs [84]—likely due to a higher perceived relevance of this kind of issue with respect to other security flaws. Such mechanisms enable timely detection of input validation bugs, possibly explaining why this kind of vulnerability is removed earlier than others. Things start to be different for the number of changes (Fig. 2b). Here too, the input validation vulnerabilities survive for fewer commits compared to the general trend. However, there is also a noticeable deviation for CWE-416 ('Use After Free') and CWE-119 ('Improper Restriction of Operations within the Bounds of a Memory Buffer'), both concerning memory management issues. Fig. 3, instead, focuses on the programming language of the project affected by the vulnerabilities. In terms of days (Fig. 3a), almost all the languages follow the general trend, with the exception for TypeScript vulnerabilities that appear to be quite isolated—in any case, the number of TypeScript vulnerabilities was too low to derive any relevant conclusion. Much more interesting is the survivability in C projects, which dominates the *'Overall'* trend. All of these observations apply to the number of changes (Fig. 3b) as well. An explanation behind this phenomenon could be the same as the high rate of survivability of memory management flaws, which could be due to the strong presence of vulnerabilities coming from the Linux kernel project, characterized by a high rate of contributions. In this regard, we separately analyzed the survival models of Linux (whose plots are available in our online appendix [27]), finding that the trend in terms of days is in line with the C-language model, whereas the number of changes' model exhibit longer survivability, possibly due to the development process adopted by the Linux community.

As a final step of our analyses, we asked whether the survival period may be related to the projects' popularity, following the idea that a larger user base may accelerate the vulnerability discovery and fixing times. We correlated the survival period of all the vulnerabilities—using both the number of days and changes—with the number of stars assigned onto the GitHub repository [85]—which is a widely used proxy metric to estimate the popularity of a project. Our analysis, done by using Spearman's rank coefficient test [62], revealed weak correlation values (both $\rho^2 < 0.1$). This result, however, does not let us derive any relevant conclusion, hence requiring further investigations involving additional contributing factors and statistical analyses.

Main findings for RQ$_3$

At least half of the vulnerabilities survive for 511 days (i.e., the probability of being fixed starts increasing after more than one year from the last contributing commit), undergoing to 9 different changes. This trend, however, is highly influenced by some extreme outliers, e.g., CVE-2014-3158, which took over 20 years to be fixed, and CVE-2018-12684, requiring 1,692 commits. Input validation vulnerabilities are fixed earlier than others, while memory management issues persist longer in the code.

3.4 RQ$_4$: How are Known Vulnerabilities Removed From the Source Code?

The statistically significant sample of 351 (fix, CVE) pairs, belonging to 66 CWEs, involves 80 different projects. The median number of files involved in fixing commits is just 1, with 282 being the highest number of changed files within a single fix. Additionally, no fix is only made of added lines: the median number of added and deleted lines is 10 and 4, respectively. Hence, it seems that a fix mainly requires more additions than deletions, suggesting that vulnerabilities are typically caused by the lack of proper control mechanism (e.g., checking the buffer size before accessing it) rather than the presence of ill-implemented features.

In general, we found large variability in how developers remove known vulnerabilities, depending on both the intrinsic characteristics of a vulnerability and the programming language. Yet, we were able to classify the recurrent removal methods and provide a taxonomy (reported in Table 5) where we described how developers fix security flaws. The most diffused removal method is *'Sanitize*

External Input', which is naturally connected to the resolution of any code injection vulnerabilities (e.g., SQL Injection, XSS, etc.) and implements proper sanitization routines of user-supplied inputs, a very common issue in web applications. We found that 27.07% of the pairs (95/351) concern vulnerabilities fixed by applying such a method, followed by the 17.38% cases (61/351) in which *'Prevent Access Over Bounds'* was applied to solve flaws related to the memory management in low-level languages, such as C and C++. Other removal methods are applied with a considerably lower frequency, but we could delineate a general consideration: most of the methods described in Table 5 refer to small program transformations that might potentially be fully automated and used in combination with vulnerability detection approaches to cover the entire refactoring pipeline, thus substantially supporting developers when dealing with security-related issues. It is worth pointing out that our taxonomy describes the removal methods which have been used in practice to tackle real software vulnerabilities. In this sense, it represents, to the best of our knowledge, the first attempt to provide an empirically grounded list of guidelines to support researchers in devising automated tools for vulnerability removal. Although existing mitigation strategies—such as the one collected by CWE—contains richer information and tips to avoid introducing weak code, they happen to be general and sometimes difficult to apply in real-world scenarios. As a further assessment, we drew connections between our taxonomy and the *"Potential Mitigations"* reported by the 66 CWEs associated with the sampled 351 (fix, CVE) pairs (Table 5). In most cases, CWE already provides general guidelines to deal with each vulnerability type appearing in our sample; however, we observed some recurrent removal methods that could not be mapped into any of the potential mitigations reported in the 66 CWEs. On the one hand, this could be caused by a limitation of our sample, as it only pertains to a limited portion of the entire CWE—indeed, *'Limit Attempts'*, *'Improve Session Management'*, and *'Avoid Deserialization of Untrusted and Harmful Data'* could be mapped to the strategies described in CWE-307, CWE-384, and CWE-502, which did not appear in our sample. On the other hand, *'Fix Initialization'*, *'Remove Vulnerable Code'*, and *'Improve Security Configuration'* appears to have no counterpart within CWE. Thus, our results may improve the current knowledge base on the matter with additional mechanisms to deal with software vulnerabilities.

Zooming into the *CWE Top 10*, Table 6 reports the recurrent methods for removing known vulnerabilities, along with their frequencies. These results are perfectly in line with the more general findings discussed so far. On the one hand, the *'Sanitize External Input'* method is the preferred choice for fixing CWE-79 ('Cross-site Scripting'), CWE-20 ('Improper Input Validation'), CWE-89 ('SQL Injection'), and CWE-78 ('OS Command Injection') vulnerabilities. On the other hand, *'Prevent Access Over Bounds'* is done to fix memory managements flaws (CWE-787, CWE-125, and CWE-119), but also, surprisingly, CWE-200 ('Exposure of Sensitive Information to an Unauthorized Actor'), which is done to avoid leaking sensitive data using buffer overflows. Both these removal methods consist in adding additional code (e.g., `if` or `assert` statements) to prevent (i) variable to receive malicious values, and (ii) going over the intended limits of buffers or files. In the end, we can confirm the feasibility of automated mechanisms able to fix them.

Main findings for RQ_4

The fixing commit generally involves one single file on which, on a median, the number of added lines is twice the deleted one. *'Sanitize External Input'* is the most recurrent method for fixing input validation issues. In most cases, such fixes consist of simple program transformations, e.g., the addition of missing checks in the source code, even when plenty of files and lines of code are involved.

4 Discussion and Implications

The results of our four research questions provided several insights that need to be discussed, as well as 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 a median, after 2 commits) and while maintaining the source code, e.g., when enhancing existing features, in line with what was already experienced by Meneely *et al.* [26]. On the one hand, our findings have implications for future research on automated solutions aiming at finding vulnerabilities in source code: we argue that novel techniques can be *time- and context-dependent*, i.e., analyzing the timeline as well as the intended actions of developers of newly committed source code files, could be devised and experimented. On the other hand, our results possibly indicate the need for additional investments on the educational side of software maintenance and evolution, especially on how to comprehend source code and diagnose its potential issues when a new change request comes in, as well as how to evolve legacy systems while keeping vulnerabilities into account, e.g., by teaching security patterns and tools to control for vulnerabilities when enhancing existing features.

More Research on Vulnerabilities is Needed. Our study constitutes a large-scale analysis of the software engineering life cycle of vulnerabilities. Yet, we still have no compelling empirical evidence on the reasons why vulnerabilities remain in a software system for such a long time. Similarly, we are still unaware of the motivations why certain actions, e.g., bug fixing operations, are more prone to introduce vulnerabilities: for example, this may be due to the lack of proper control mechanisms, a lack of knowledge of basic security principles, or other undocumented reasons. All these aspects represent challenges for the software engineering research community. We hope that these results can stimulate a more comprehensive investigation into software vulnerabilities and security practices in general.

Increasing the Developer's Awareness. The survival analyses conducted in the context of $\mathbf{RQ_3}$ revealed two notable findings. On the one hand, they suggest a general lack of awareness with respect to the presence of vulnerabilities. This represents a call for researchers working on the intersection

TABLE 5
The Taxonomy of the Recurrent Methods Used to Remove Vulnerabilities

Method	Description	CWE Mapping
	Users and Networking	
Hide Sensitive Information	Do not allow passwords/tokens/keys to be given to users (e.g., via outputs, stack traces, or source code).	Unnamed "Architecture and Design" Strategies at CWE-798
Change User Permissions	Change the assignment of either intended or unintended functionalities or ports to the different users/actors.	"Separation of Privilege" Strategy
Block Untrusted Hosts	Remove or block hosts considered untrusted.	"Separation of Privilege" Strategy (partially)
Limit Attempts	Limit the attempts to sensitive functionalities, such as logins or payments.	N/A
Improve Certificate Verification	Check the certificate validity (its digital signature and origin).	Unnamed "Implementation" Strategy at CWE-295
Improve Session Management	Set a session timeout, invalidate the session on logout, or reuse a pre-existing session only when needed.	N/A
Ask User Confirmation	Add a confirm dialog or an extra input to avoid issuing operations unintentionally.	Unnamed "Architecture and Design" Strategy at CWE-352
	Memory Management	
Prevent Access Over Bounds	Ensure a pointer's offset does not go beyond the buffer limit or check if the signed integer buffer's length does not overflow, e.g., when used in `fread()`.	Unnamed "Implementation" Strategy at CWE-119
Check Before Dereferencing	Check if a pointer is not NULL before dereferencing it, or if a file is not a symlink.	Unnamed "Implementation" Strategy at CWE-119, CWE-476
Check Input Size	Check the size (bytes) of externally supplied data before placing it in a buffer.	"Input Validation" Strategy
Change Buffer Size	Increase/decrease the size of a dynamically allocated buffer (e.g., `malloc()`-like functions) or statically allocated variable.	"Input Validation" Strategy
Improve Resource Management	Set a limit to the number of allocated resources (e.g., TCP sack holes, file descriptors, etc.); avoid opening them when not strictly needed or close them when not needed anymore; cleanup memory leaks.	Unnamed"Architecture and Design" Strategy at CWE-400, CWE-401, CWE-770
Remove Invalid Free	Remove `free()` calls on non-dynamically allocated buffers or avoid doing double free.	Unnamed "Input Validation" at CWE-415
Set Pointer to Null After Free	Set a pointer to NULL after its use in a `free()`.	Unnamed "Implementation" Strategy at CWE-416
Remove Type Confusion	Remove cases where a resource (e.g., a buffer) is accessed improperly (e.g., with unions or wrong pointer types).	Unnamed "Implementation" Strategy at CWE-20
	Implementation	
Sanitize External Input	Add or improve checks on the content of externally supplied data, either rejecting it or adjusting the content (e.g., escaping/encoding special characters, remove decimals, relying on parameterization).	"Input Validation" Strategy
Handle Error Cases	Handle exceptions/signals/error return values previously not considered. This comprises loop exit conditions, division-by-zero, and integer under/overflows, as well.	Unnamed "Implementation" Strategy at CWE-190, CWE-252 (partially)
Fix Initialization	Put objects/structs/buffers in a proper initial state, e.g., when creating an object starting from another (clone).	N/A
Employ New Algorithm	Employ a brand new algorithm/mechanism for checking security-related aspects (e.g., CSRF protection), either from scratch or third-party solutions.	Unnamed "Architecture and Design" Strategies at CWE-352 (partially)
Remove Vulnerable Code	Remove the vulnerable feature/file code completely.	N/A
Remove Race Conditions	Fix concurrency issues, e.g., improving the locks management.	Unnamed "Implementation" Strategy at CWE-362
Avoid Deserialization of Untrusted and Harmful Data	Deserialize only the unharmful part of untrusted data or prevent it at all.	N/A
	Configuration	
Improve Security Configuration	Modify security-related configuration (e.g., framework settings, configuration files, dependencies).	N/A

The table also reports the overlap with the Potential Mitigations *related to the weakness types of the 351 (fix, CVE) pairs sampled in* $\boldsymbol{RQ}_4$.

TABLE 6
The Most Recurrent Removal Methods With Respect to the *CWE Top 10* Observed in the Sample Used for $\mathbf{RQ_4}$

CWE	Most Recurrent Method	Occurrences	Percentage
CWE-79	Sanitize External Input	41/45	91.11%
CWE-787	Prevent Access Over Bounds	6/9	66.67%
CWE-20	Sanitize External Input	13/31	41.94%
CWE-125	Prevent Access Over Bounds	11/26	42.31%
CWE-119	Prevent Access Over Bounds	15/33	45.46%
CWE-89	Sanitize External Input	7/10	70.00%
CWE-200	Prevent Access Over Bounds	5/18	27.78%
CWE-416	Check Input Size	2/7	28.57%
CWE-352	Employ New Algorithm	2/3	66.67%
CWE-78	Sanitize External Input	4/5	80.00%

between software maintenance and human aspects: it would indeed be worth exploring novel methods able to increase the developer's awareness about security issues (e.g., through summarization or visualization of vulnerability-related information). On the other hand, we noticed that vulnerabilities related to input validation issues tend to have a shorter life cycle, possibly indicating that these can be quickly identified or might be considered as more harmful—opposed to memory management issues characterizing low-level programming languages like C or C++. In this respect, further research around the topic of vulnerability prioritization, driven by the developer's perceived harmfulness, might greatly increase the ability of developers to deal with security issues—as shown by recent work targeting design flaws [86]. Researchers are also encouraged to delve into the effects that highly popular projects have on vulnerability discovery and fixing times.

On Security Testing and More. Our results indicate that the majority of vulnerabilities remain in a system for a long while. Besides the lack of awareness discussed above, this finding also suggests that developers do not have proper mechanisms to verify source code for the presence of vulnerabilities. Hence, we argue that the definition of advanced verification mechanisms able to assist developers when looking for possible security issues should be devised: this represents a further challenge for the entire software testing community and, perhaps, even for automatic test case generation [87], [88]. In addition, enabling alternative strategies—e.g., novel code review practices for vulnerabilities—could be an interesting path to allow developers to better spot errors statically [89].

On Human Aspects and Vulnerabilities. As also suggested by previous work [74], [90], [91], human factors play a role also when considering the introduction of vulnerabilities. At first, most security issues are introduced by developers showing a high workload. Besides, both expert developers and newcomers have roughly the same chances to incur security issues. These results call for a more comprehensive overview of software vulnerability research that includes human-related considerations: for instance, novel methods to triage developers' work 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 [60], [92], influence the introduction of security issues. Lastly, the Human Error Theory defined by Reason [93] may give additional insights on what kind of errors developers make when introducing defects such as vulnerabilities, as investigated by Nagaria and Hall [94], who discovered that such errors could be related to the complex development environments in which developers write their code, as well as a lack of focus and concentration. Moreover, the severity of the flaws (e.g., measured using the CVSS score) should not be neglected when investigating the reasons behind the introduction of software vulnerabilities.

Automating the Removal Process. Other valuable results from our empirical study concern the study of how software vulnerabilities are removed. From $\mathbf{RQ_4}$ we could establish a taxonomy of removal operations performed by developers when fixing vulnerabilities and discovered that most of these actions are simple to program transformations (e.g., escaping functions) that have the potential to be fully 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.

All these implications represent challenges that we aim at facing as part of our future research agenda on the matter.

5 Threats to Validity

This section discusses the limitations that may have influenced our findings and how we mitigated them.

Construct Validity. Our results could be affected by the wrong identification of both vulnerability-fixing patches and vulnerability-contributing commits. As for the former, we relied on the vulnerability fixing commits available in NVD: while this database is curated and improved constantly, we cannot exclude that a patch may not remove the vulnerability as intended. Indeed, we discovered that some projects regularly make tangled fixing commits [55], which mixes both vulnerability patches with other ordinary development or maintenance activities. Such fixes, not only are difficult to comprehend, but they also damage the performance of automated mining techniques based on them, such as our VCCs mining procedure. As for the latter, we relied on a technique based on the SZZ algorithm, which analyzes the previous history of a fixed file to identify the commit(s) that likely introduced the vulnerability. Previous studies have shown that this algorithm may frequently produce false positives [52]; to account for this aspect and control the reliability of SZZ in our context, we took some precautions. First, we employed PyDriller, a tool

implementing a standard version of the algorithm on which, however, we added some additional adjustments to reduce the number of false positives, e.g., discarding the VCCs where only comments, cosmetic changes, or empty lines are blamed. In particular, we adopted some filters to discard those files not representing source code, such as documentation, build or test files. We employed regular expressions to filter our those files not fulfilling the most common naming conventions (e.g., detecting test files whether they begin or end with the *'test'* substring). However, this solution could have missed some exceptional cases (e.g., a test file not having the *'test'* substring in its name), which might have contributed negatively to the performance of our VCCs mining validation. Second, we manually re-assessed the effectiveness of the algorithm for a sample of 376 pairs (CVE, VCC) in our dataset, finding a precision of 68%. Aware of the fact that such performance may have affected the results of our analyses, we re-run the entire analysis pipeline onto the pairs (CVE, VCC) labeled as correct by the inspectors. The results we obtained are in line with the general ones, raising our confidence in the employed technique—the raw data of this sanity check were uploaded into our online appendix [27]. Finally, the `git-blame` functionality automatically recognizes file renamings when traversing the history, so we further reduce the risk of blaming incorrect commits. Specifically, when a commit changes the path of a file but keeps intact its whole content, `git` easily identifies the renaming. Should a commit apply a renaming alongside other code changes, `git` would still be able to recognize the fact that the file is not a new one. Internally, `git` compares the renamed file contents and computes a similarity index—based on the number of changed lines compared to the file's size. By default, `git` uses a threshold of 0.5, meaning that the new file is considered a renaming of a pre-existing one if they are at least 50% similar.[6] Although this threshold could be freely set by the user, we left it to 0.5 as it is the default choice of diff views, such as the one shown by GitHub.[7]

Current mining approaches based on SZZ still suffer from the cases in which the commit is made only of added lines [50]. Our algorithm, as well as the ones adopted in [95], [96], handled these cases by blaming the contextual lines as well, despite in a different way. We have also avoided blaming the contextual lines of change blocks made only of new functions or methods. Yet, we could not apply it thoroughly, as we were limited by the parsing capabilities offered by Lizard library. The work by Sahal and Tosun [97] proposes an approach to consider the commits that previously changed the code block (e.g., the statements of a `for` loop) that contained the continuous block of added lines. We decided not to adopt this approach as it has not been evaluated extensively, and it required parsing source code files of many different programming languages in order to map the change blocks to the belonging code block element.

It is worth noting that, despite the metrics to compute the developers' *workload* have been used in the past [22], [60], these represent only a proxy to the actual amount of work done, as commits may require different amounts of work, and developers may work on various projects.

The *Working on Release* metric uses absolute time references to determine the proximity to a deadline, ranging from *'the same day of'* to *'over one month before'* a release. We recognize the fact that this choice does not take into account *project-specific release cycles*—i.e., the number of days elapsing between the major releases of a project—which would better describe the actual release process. However, to the best of our knowledge, there is no tool able to infer the actual release cycle of a given project only by looking at the dates of the releases. Any approximate solution—such as computing the average of days elapsed between releases—would have introduced even more biases in our observations. Hence, we opted for a simple and general implementation of this concept that suits our large context, as was done by Vassallo *et al.* [57], leaving additional interpretation of the results to the reader.

Internal Validity. To address $\mathbf{RQ}_2$, we implemented a script to automatically characterize VCCs. To study the *'commit goal'* category, we re-implemented a previously proposed keyword-matching approach [98], [99], which has been shown to be highly accurate [56]. As for the other categories, we re-implemented previously proposed algorithms [22], [59], [60] following the exact description reported in those papers; despite the extensive testing phase, we cannot exclude possible implementation errors. For the sake of replicability, we made all data and scripts employed publicly available [27]. A particularly interesting point to comment upon is connected to the fact that a number of VCCs were tagged as both refactoring and other operations (i.e., new feature, bug fixing, or enhancement). This reflects what has been reported by previous research: developers tend to perform *floss refactoring* actions [57], [100], [101], that is, the application of refactoring operations during other maintenance and evolution activities. The keyword-matching approach could actually identify multiple actions that developers normally perform in practice. This provides further confidence in the validity of the approach.

Conclusion Validity. In addressing our research questions, we provided some insights coming from outliers, such as vulnerabilities whose introduction was caused by many contributing commits ($\mathbf{RQ}_1$). While these data points still let us distill some valuable findings, we could have been subject to possible wrong inteprations of the statistics extracted from the datasets employed in each **RQ**. For this reason, we re-ran the entire analysis pipeline without considering the extreme outliers, observing that these new results—which can be found in our online appendix [27]—were in line with the ones observed during the main study.

In the context of $\mathbf{RQ}_2$, we characterized VCCs by considering a number of aspects, i.e., commit goal, project-, and developer-status. While this allowed us to contextualize the contributions to the introduction of vulnerabilities, it is important to point out that recent work [102] has shown that not all defects are due to development activities performed within a project and that, instead, some defects are inherited by third-party libraries. This might be potentially true in our case as well. Nevertheless, there are three key observations to be done in this respect. First, the vulnerability types considered in our study are *by definition* due to errors when developing source code: for instance, if we consider the *CWE Top 10* presented in Table 1, it is pretty straightforward to observe that all of them are due to improper control mechanisms implemented by developers. As such, the influence of the so-called *extrinsic*

6. https://git-scm.com/docs/git-diff
7. https://github.blog/2010-01-12-improved-commit-diffs/

defects on our results must be necessarily limited. In the second place, it is worth remarking that CWE marks such inherited defects with a special category, i.e., *'Using Components with Known Vulnerabilities'*: our dataset does not include this type of issues, hence not being affected by extrinsic vulnerabilities. Of course, there would still be the case that some vulnerabilities in our dataset might have been mislabeled. While we cannot exclude this possibility, (1) we argue that this is quite unlikely given the accuracy with which NVD is managed; and (2) a precise estimation of how many vulnerabilities are mislabeled is not practically feasible since this analysis would require the re-compilation of the considered commits: besides being time-consuming, this analysis might be notably biased by the presence of broken snapshots [103].

We reported our findings under the perspective of the *CWE Top 10 Most Dangerous Software Weaknesses*. Although the CWE taxonomy is a tree made of weakness types with parent-children relationships, the *CWE Top 10* reports a list of CWEs at different abstraction levels. For example, CWE-787 (*'Out-of-bounds Write'*) and CWE-119 (*'Improper Restriction of Operations within the Bounds of a Memory Buffer'*) occupy the second and fifth place of the Top 10, respectively, but CWE-787 is a subtype of CWE-119 (in other words, CWE-119 is labeled as 'class' abstraction type, while CWE-787 as 'base' abstraction type). This could bias the conclusions we made on the results divided by the CWEs, as the Top 10 does not take the parenthood relationships into account. We chose not to re-classify the CVEs into many CWEs, but only consider the single CWE provided by NVD. Additional research on the matter would improve the interpretation of the results we obtained.

Survival analysis ($\mathbf{RQ}_3$) is a standard approach for investigating lifetime expectancy [104], [105], yet a possible threat concerns the metrics we used to determine the vulnerability survival, i.e., the number of days and changes from their introduction to the removal. We recognize that project activity or developers' availability may influence the two variables. In addition, it is worth remarking that the survival analysis takes into account the vulnerable period between the last vulnerability-contributing (turning point) and vulnerability-fixing commits: however, a vulnerability might have been disclosed or the related patch made available by NVD way later than the introduction or just before the removal and, for this reason, developers might not have had the entire vulnerable period to discover a vulnerability and apply the fix. Our $\mathbf{RQ}_3$ only takes the change history of the considered projects into account, without considering external events that might have influenced the vulnerability identification and fixing process; nevertheless, we argue that our findings are still extremely relevant because they report on the lifespan of vulnerabilities in the considered projects, presenting the period in which they were actually exploitable by external attackers. Of course, replications that consider external events would complement our findings.

Concerning $\mathbf{RQ}_4$, we conducted a manual investigation to determine how vulnerabilities are removed: more inspectors were involved in the process to increase the accuracy of the classification of the fixing strategies. Nevertheless, we cannot exclude imprecision and subjectiveness. Replications are, therefore, still desirable.

External Validity. We considered 3,663 of 144 different types coming from 1,096 open-source projects. As such, the size of our dataset is comparable with other large-scale software engineering studies (e.g., [22], [29]). However, the life cycle of vulnerabilities affecting other systems (e.g., developed in closed-source settings) may differ. Still, it is important to point out that our study does not cover all instances of vulnerabilities affecting the subject systems, but rather it is limited to those for which a fixing commit was available. Similarly, we are aware that not all vulnerabilities may have been reported to the CVE (especially if discovered internally by a developer and not publicly disclosed). On the one hand, the large-scale nature of our study makes us confident that similar results would have been achieved when considering the vulnerabilities that were missed in our analyses. On the other hand, we highlight that replications of our study would be desirable as they may provide additional insights into the life cycle of software vulnerabilities.

6 Related Work

Our study aims at integrating the current knowledge on the life cycle of software vulnerabilities. One of the first works that shed light on this topic is the one by Meneely *et al.* [26], who analyzed and described the properties of vulnerability-contributing commits. The authors traced 68 vulnerability reports of Apache HTTP Server—obtained from different data sources—to the VCCs over the entire history of the software system. The mined VCCs were investigated predominantly from a quantitative point of view, focusing on (i) the size of changes in terms of various code churn metrics, (ii) the amount of changed code that was previously added by different contributors (i.e., interactive churn), (iii) the author's expertise with respect to the modified files, (iv) the vulnerability exposure time in terms of days and commits, (v) the file's past vulnerability rate, and (vi) the community dissemination via release notes. The main results showed that VCCs are at least twice larger than non-VCCs, and are more likely to be done by developers touching code parts of which they have no prior knowledge. Moreover, vulnerabilities are regularly spread across the entire project history, i.e., there are no specific project phases in which vulnerabilities tend to appear more or less. This work also led to the creation of Vulnerability History Project,[8] which aims at keeping a curated dataset of accurate descriptions of the entire life cycle of known software vulnerabilities. These results integrate our findings, with the only exception to the expertise of developers that contributes to the introduction of a vulnerability. This may be explained by the fact that Meneely *et al.* classified as 'unexperienced' a developer who committed to files on which she never wrote a single line, while our study relies on the number of months elapsed since the developer's first contribution to the project. In order to detect the VCCs, the author relied on a manual code inspection methodology driven by the `git-bisect` functionality. In summary, given a vulnerability: (1) identify its fixing commits from the collected vulnerability reports, (2) write an ad-hoc vulnerability detection script that statically searches for the vulnerable code snippets in the code version before the fix, (3) run `git-bisect`, which

8. http://vulnerabilityhistory.org/

runs a binary search of all the commits before the fix, and launches the detection script at each step, gradually narrowing down the search space. The commits returned by `git-bisect` at each round were manually inspected, and corrections to the detection script were applied when needed to reduce the risk of false positives. The main difference between this methodology and our VCCs extraction algorithm is that our proposed solution is a fully automated solution that starts from the collection of vulnerability reports from NVD to the estimated set of VCCs. Another difference in our work lies in the way we analyze the properties of VCCs. Indeed, our goal was not to delve into the characteristics of VCCs alone, but rather give a general overview of the entire life cycle of software vulnerabilities, from their introduction to their resolution in the source code. For this reason, we have left out the comparison with non-VCCs, as it would have been out of the scope of this paper.

Other studies have proposed automated techniques to detect vulnerability-contributing commits. Perl *et al.* [95] proposed a machine learning-based approach, named VCCFinder, to automatically classify commits as suspicious vulnerability-contributing, based on several process software metrics. The dataset they employed to train and evaluate the model was made of 170,860 commits coming from 66 C/C++ projects. The labeling of the commits as either VCCs or non-VCCs they adopted was based on an automated technique that fetches the VCCs of a vulnerability starting from its fixing commits reported in the vulnerability report. They implemented a set of heuristics based on the results returned by the `git-blame` command, which was run on (1) each of the deleted lines in the commit, and (2) blame the lines before and after continuous blocks of code made only of added line. The authors also provided a manual evaluation of the 15% sample of all VCCs flagged by their heuristics, scoring an error rate of 3.1%. These heuristics share many similarities with the ones we have adopted in this work. The main differences lie in the corrective factors: we excluded empty lines and lines containing comments, as well as irrelevant files, such as documentation files. Moreover, we did not blame the contextual lines of blocks of added lines when they were made only of new functions or methods, as they could be inserted anywhere in the code. These corrections were also adopted in the heuristics proposed in the work by Yang *et al.* [96], which currently represent the state-of-the-practice of automatic VCCs detection. Their manual validation of 112 (fixing commit, VCC) pairs scored an error rate of 4.5%. Their algorithm, however, returns only the VCC that has been blamed most during the various runs of `git-blame` for a specific vulnerability. Our work deliberately ignored this choice, as returning only a single VCC would have underestimated the actual set of VCCs.

Morrison *et al.* [24] investigated the differences between vulnerabilities and non-security defects in source code, finding the former to be introduced later in the development process and fixed by means of different mechanisms. They extended the well-known Orthogonal Defect Classification [106], a scheme to classify defect data, proposing the ODC + Vulnerabilities (ODC+V) taxonomy. Our study complements on a large-scale the Morrison *et al.*'s work [24] not only by enlarging the available body of knowledge on vulnerabilities through a systematic analysis of how they are introduced and removed but also by presenting additional insights on their longevity and the circumstances around their introduction: these can be more pragmatically employed by the research community to device novel instruments to deal with vulnerabilities.

Canfora *et al.* [107] studied the vulnerability fixing process, comparing it with the one adopted for traditional defects. Their findings reported that vulnerabilities typically require more re-assignments in the triage phase and specific skills to be fixed. Our study does not focus on the fixing process, but rather on the entire vulnerability life cycle, from the introduction to removal. However, the results of our study—especially those of $\mathbf{RQ}_2$—corroborate and enlarge the findings by Canfora *et al.* [107] on the importance of human factors for the management of vulnerabilities.

Shahzad *et al.* [23] conducted an exploratory study to investigate the life cycle of 46,310 vulnerabilities. Similarly to our study, the authors investigated the various phases of the life cycle, e.g., the evolution of vulnerabilities during the years. The main findings of the paper showed that the percentage of remotely exploitable vulnerabilities has increased over the years and that, on average, the time taken by hackers to exploit a vulnerability is lower than the one taken by vendors. With respect to this paper, our work has a different *granularity*: Shahzad *et al.* [23] considered that the life cycle of a vulnerability starts when it is discovered by the vendor, a hacker, or any third-party software analyst and ends when all users of the software install the vulnerability fixing patch; our work, instead, considers as life cycle the period between the vulnerability introduction into and its removal from the code. In other words, while the previous work aimed at providing insights from a more managerial perspective, we lowered the level and considered the life cycle of a vulnerability within software repositories with the goal of suggesting possible improvements in the tools made available to developers. As such, the types of analyses performed and the methodological steps conducted are necessarily different and provide complementary findings to those reported by Shahzad *et al.* [23].

Smith *et al.* [108] investigated how developers use security-focused static analyses to solve security defects, finding that to effectively use them developers would like to have contextual information (e.g., classes possibly presenting side-effects). These findings have been later confirmed by Muniz *et al.* [20]. In this sense, our paper can actually help understanding which circumstances lead to the introduction of vulnerabilities, possibly informing developers of existing tools on how to create a context while detecting them. Besides the empirical studies, it is also worth mentioning the existence of a large amount of papers targeting the definition of automated solution to identify and remove vulnerabilities. Previous work [13], [14], [15], [16] has indeed evaluated a number of security tools, reporting that most of them suffer from false positives. Clearly, our work cannot be compared to these ones, yet our empirical findings might again be useful to improve existing techniques.

7 Conclusion

We presented a large-scale empirical study conducted over the life cycle of 3,663 of 144 weakness types belonging to 1,096 open-source projects. We aimed at understanding how and under which circumstances vulnerabilities are

contributed to, what is their survivability, and how they are removed. These results provide several findings: (i) contributions to vulnerabilities are made shortly after the creation of new files; (ii) developers contributes to vulnerabilities while maintaining source code; (iii) developers with higher workload contribute to most security flaws; (iv) vulnerabilities have high survivability rates in terms of days they stay in the source code. Based on those findings, we presented a set of implications and open challenges that the research community should embrace to improve the support given to practitioners when identifying and fixing software vulnerabilities in practice. The set of challenges identified are all part of our future agenda, which will aim at investigating in greater detail the phenomenon of software vulnerabilities, how to detect them automatically, and how certain development theories (e.g., the Human Error Theory [93]) apply to the problem. All these findings we presented should be compared and related to the ones already reported into known vulnerability repositories, such as the one curated by the VULNERABILITY HISTORY PROJECT.[8] Indeed, we plan to design an automated tool that periodically mines data from the National Vulnerability Database, processes them, and subsequently contributes to the VULNERABILITY HISTORY PROJECT. We also plan to integrate our empirical findings—particularly the ones regarding the vulnerability removal methods ($\mathbf{RQ}_4$)—into CWE, which accepts content suggestions to further improve the quality of its knowledge base. Moreover, there is need to improve the performance of VCCs mining heuristics with additional experimentations to (i) understand what makes the mining algorithms prone to errors, and (ii) learn to automatically remove the noise from vulnerability fixes. Finally, the relations between the findings we observed, e.g., the high rate of newly-created files in VCCs ($\mathbf{RQ}_1$) and their pronesses to remove bugs ($\mathbf{RQ}_2$), deserve additional investigations to understand how traditional and security bugs are fixed by developers [107].

ACKNOWLEDGMENTS

The authors would like to sincerely thank the Associate Editor and anonymous Reviewers for the insightful comments and feedback provided during the review process.

REFERENCES

[1] R. Shirey, "Internet security glossary, version 2," 2007. Accessed: Jan. 14, 2022. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc4949

[2] S. Frei, D. Schatzmann, B. Plattner, and B. Trammell, *Modeling the Security Ecosystem - the Dynamics of (In)Security*. Berlin, Germany: Springer, 2010, pp. 79–106.

[3] M. Finifter, D. Akhawe, and D. Wagner, "An empirical study of vulnerability rewards programs," in *Proc. USENIX Conf. Secur.*, 2013, pp. 273–288.

[4] C. Pfleeger and S. Pfleeger, *Security in Computing*. Englewood Cliffs, NJ, USA: Prentice-Hall, 2002.

[5] Wannacry ransomware attack, 2017. [Online]. Available: https://en.wikipedia.org/wiki/WannaCry_ransomware_attack

[6] Mediacenter, 2018. [Online]. Available: https://www.pandasecurity.com/mediacenter/security/consequences-not-applying-patches/

[7] D. Aranha, P. Barbosa, T. Cardoso, C. Araújo, and P. Matias, "The return of software vulnerabilities in the brazilian voting machine," *Comput. Secur.*, vol. 86, pp. 335–349, 2019.

[8] M. Castro, M. Costa, and T. Harris, "Securing software by enforcing data-flow integrity," in *Proc. Symp. Oper. Syst. Des. Implementation*, 2006, pp. 147–160.

[9] S. Ardi, D. Byers, P. Meland, I. Tondel, and N. Shahmehri, "How can the developer benefit from security modeling?," in *Proc. Int. Conf. Availability Rel. Secur.*, 2007, pp. 1017–1025.

[10] S. Mirhosseini and C. Parnin, "Can automated pull requests encourage software developers to upgrade out-of-date dependencies?," in *Proc. Int. Conf. Automated Softw. Eng.*, 2017, pp. 84–94.

[11] G. McGraw, *Software Security: Building Security In*. Reading, MA, USA: Addison-Wesley, 2006.

[12] P. Morrison, B. Smith, and L. Williams, "Surveying security practice adherence in software development," in *Proc. Hot Top. Sci. Secur. Symp. Bootcamp*, 2017, pp. 85–94.

[13] A. Austin and L. Williams, "One technique is not enough: A comparison of vulnerability discovery techniques," in *Proc. Int. Symp. Empir. Softw. Eng. Meas.*, 2011, pp. 97–106.

[14] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A static analysis tool for detecting web application vulnerabilities," in *Proc. Symp. Secur. Privacy*, 2006, pp. 258–263.

[15] V. B. Livshits and M. S. Lam, "Finding security vulnerabilities in java applications with static analysis," in *Proc. Conf. USENIX Secur. Symp.*, 2005, pp. 18–18.

[16] M. Martin, B. Livshits, and M. Lam, "Finding application errors and security flaws using PQL: A program query language," in *proc. Conf. Object-Oriented Program. Syst. Lang. Appl.*, 2005, pp. 365–383.

[17] I. Chowdhury and M. Zulkernine, "Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities," *J. Syst. Archit.*, vol. 57, no. 3, pp. 294–313, 2011.

[18] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, "Predicting vulnerable software components," in *Proc. Conf. Comput. Commun. Secur.*, 2007, pp. 529–540.

[19] F. Li and V. Paxson, "A large-scale empirical study of security patches," in *Proc. Conf. Comput. Commun. Secur.*, 2017, pp. 2201–2215.

[20] 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 *Proc. Int. Workshop Variability Model. Softw.-Intensive Syst.*, 2018, pp. 51–58.

[21] M. di Biase, M. Bruntink, and A. Bacchelli, "A security perspective on code review: The case of chromium," in *Proc. Int. Work. Conf. Source Code Anal. Manipulation*, 2016, pp. 21–30.

[22] M. Tufano *et al.*, "When and why your code starts to smell bad (and whether the smells go away)," *IEEE Trans. Softw. Eng.*, vol. 43, no. 11, pp. 1063–1088, Nov. 2017.

[23] M. Shahzad, M. Z. Shafiq, and A. X. Liu, "A large scale exploratory analysis of software vulnerability life cycles," in *Proc. Int. Conf. Softw. Eng.*, 2012, pp. 771–781.

[24] P. Morrison, R. Pandita, X. Xiao, R. Chillarege, and L. Williams, "Are vulnerabilities discovered and resolved like other defects?," *Empir. Softw. Eng.*, vol. 23, no. 3, pp. 1383–1421, 2018.

[25] National vulnerability database, 2022. [Online]. Available: https://nvd.nist.gov/

[26] A. Meneely, H. Srinivasan, A. Musa, A. Tejeda, M. Mokary, and B. Spates, "When a patch goes bad: Exploring the properties of vulnerability-contributing commits," in *Proc. Int. Symp. Empir. Softw. Eng. Meas.*, 2013, pp. 65–74.

[27] E. Iannone, R. Guadagni, F. Ferrucci, A. De Lucia, and F. Palomba, The secret life of software vulnerabilities: A large-scale empirical study - online appendix, 2022. [Online]. Available: https://github.com/sesalab/OnlineAppendices/tree/main/TSE21-VulnerabilityLifecycle

[28] L. Pascarella, F. Palomba, and A. Bacchelli, "Fine-grained just-in-time defect prediction," *J. Syst. Softw.*, vol. 150, pp. 22–36, 2019.

[29] Y. Kamei *et al.*, "A large-scale empirical study of just-in-time quality assurance," *IEEE Trans. Softw. Eng.*, vol. 39, no. 6, pp. 757–773, Jun. 2013.

[30] N. Sae-Lim, S. Hayashi, and M. Saeki, "Context-based code smells prioritization for prefactoring," in *Proc. Int. Conf. Program Comprehension*, 2016, pp. 1–10.

[31] B. Kitchenham, "Evaluating software engineering methods and tool part 1: The evaluation context and evaluation methods," *Softw. Eng. Notes*, vol. 21, no. 1, pp. 11–14, 1996.

[32] B. A. Kitchenham, T. Dyba, and M. Jorgensen, "Evidence-based software engineering," in *Proc. Int. Conf. Softw. Eng.*, 2004, pp. 273–281.

[33] T. Menzies *et al.*, "Local versus global lessons for defect prediction and effort estimation," *IEEE Trans. Softw. Eng.*, vol. 39, no. 6, pp. 822–834, Jun. 2013.

[34] R. Miller, *Survival Analysis*. Hoboken, NJ, USA: Wiley, 2011.

[35] N. Baddoo and T. Hall, "De-motivators for software process improvement: An analysis of practitioners' views," *J. Syst. Softw.*, vol. 66, no. 1, pp. 23–33, 2003.

[36] G. Canfora and L. Cerulo, "Impact analysis by mining software and change request repositories," in *Proc. Int. Softw. Metrics Symp.*, 2005, pp. 9–29.

[37] G. Bavota, A. De Lucia, M. Di Penta, R. Oliveto, and F. Palomba, "An experimental investigation on the innate relationship between quality and refactoring," *J. Syst. Softw.*, vol. 107, pp. 1–14, 2015.

[38] B. Du Bois, S. Demeyer, and J. Verelst, "Refactoring-improving coupling and cohesion of existing code," in *Proc. Work. Conf. Reverse Eng.*, 2004, pp. 144–151.

[39] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Reading, MA, USA: Addison-Wesley, 1999.

[40] U.S. NIST computer security division, 2022. [Online]. Available: https://www.nist.gov

[41] Common vulnerabilities and exposures, 2022. [Online]. Available: https://cve.mitre.org/

[42] O. Alhazmi, Y. Malaiya, and I. Ray, "Measuring, analyzing and predicting security vulnerabilities in software systems," *Comput. Secur.*, vol. 26, no. 3, pp. 219–228, 2007.

[43] P. Mell, K. Scarfone, and S. Romanosky, "Common vulnerability scoring system," *Secur. Privacy*, vol. 4, no. 6, pp. 85–89, 2006.

[44] S. Huang, H. Tang, M. Zhang, and J. Tian, "Text clustering on national vulnerability database," in *Proc. Int. Conf. Comput. Eng. Appl.*, 2010, pp. 295–299.

[45] S. Zhang, D. Caragea, and X. Ou, "An empirical study on using the national vulnerability database to predict software vulnerabilities," in *Proc. Int. Conf. Database Expert Syst. Appl.*, 2011, pp. 217–231.

[46] CWE top 25 most dangerous software weaknesses, 2020. [Online]. Available: https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html

[47] CVE search tool, 2022. [Online]. Available: https://github.com/cve-search/cve-search

[48] Git - git-merge documentation, 2022. [Online]. Available: https://git-scm.com/docs/git-merge

[49] J. Sliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?," in *Proc. Int. Workshop Mining Softw. Repositories*, 2005, pp. 1–5.

[50] D. 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," *IEEE Trans. Softw. Eng.*, vol. 43, no. 7, pp. 641–657, Jul. 2017.

[51] D. Spadini, M. Aniche, and A. Bacchelli, "PyDriller: Python framework for mining software repositories," in *Proc. 26th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2018, pp. 908–911.

[52] G. Rodr íguez-Pérez, G. Robles, and J. Gonz ález-Barahona, "Reproducibility and credibility in empirical software engineering: A case study based on a systematic literature review of the use of the SZZ algorithm," *Inf. Softw. Technol.*, vol. 99, pp. 164–176, 2018.

[53] J. Cohen, "A coefficient of agreement for nominal scales," *Educ. Psychol. Meas.*, vol. 20, no. 1, pp. 37–46, 1960.

[54] M. McHugh, "Interrater reliability: The kappa statistic," *Biochemia Medica : Časopis Hrvatskoga Društva Medicinskih Biokemičara / HDMB*, vol. 22, pp. 276–82, 2012.

[55] K. Herzig and A. Zeller, "The impact of tangled code changes," in *Proc. 10th Working Conf. Mining Softw. Repositories*, 2013, pp. 121–130.

[56] 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," *Trans. Softw. Eng.*, vol. 44, no. 10, pp. 977–1000, Oct. 2018.

[57] C. Vassallo, G. Grano, F. Palomba, H. Gall, and A. Bacchelli, "A large-scale empirical exploration on refactoring activities in open source software projects," *Sci. Comput. Program.*, vol. 180, pp. 1–15, 2019.

[58] S. G. K. Patro and K. K. Sahu, "Normalization: A preprocessing stage," *CoRR*, vol. abs/1503.06462, 2015. [Online]. Available: http://arxiv.org/abs/1503.06462

[59] B. Vasilescu *et al.*, "Gender and tenure diversity in GitHub teams," in *Proc. Conf. Hum. Factors Comput. Syst.*, 2015, pp. 3789–3798.

[60] 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?" *IEEE Trans. Softw. Eng.*, vol. 47, pp. 108–129, Jan. 2021.

[61] Vulnerability metrics, 2022. [Online]. Available: https://nvd.nist.gov/vuln-metrics/cvss

[62] C. Spearman, "The proof and measurement of association between two things," *Int. J. Epidemiol.*, vol. 39, no. 5, pp. 1137–1150, 2010.

[63] E. L. Kaplan and P. Meier, "Nonparametric estimation from incomplete observations," *J. Amer. Statist. Assoc.*, vol. 53, no. 282, pp. 457–481, 1958. [Online]. Available: https://www.tandfonline.com/doi/abs/10.1080/01621459.1958.10501452

[64] H. Hsieh and S. Shannon, "Three approaches to qualitative content analysis," *Qualitative Health Res.*, vol. 15, no. 9, pp. 1277–1288, 2005.

[65] B. Smith and L. Williams, "Using SQL hotspots in a prioritization heuristic for detecting all types of web application vulnerabilities," in *Proc. 4th IEEE Int. Conf. Softw. Testing Verification Valid.*, 2011, pp. 220–229.

[66] L. K. Shar, H. Beng Kuan Tan, and L. C. Briand, "Mining SQL injection and cross site scripting vulnerabilities using hybrid program analysis," in *Proc. 35th Int. Conf. Softw. Eng.*, 2013, pp. 642–651.

[67] D. Oliveira, M. Rosenthal, N. Morin, K.-C. Yeh, J. Cappos, and Y. Zhuang, "It's the psychology stupid: How heuristics explain software vulnerabilities and how priming can illuminate developer's blind spots," in *Proc. 30th Annu. Comput. Secur. Appl. Conf.*, 2014, pp. 296–305.

[68] A. Hindle, D. M. German, and R. Holt, "What do large commits tell us?: A taxonomical study of large commits," in *Proc. Int. Work. Conf. Mining Softw. Repositories*, 2008, pp. 99–108.

[69] C. Abid, M. Kessentini, V. Alizadeh, M. Dhouadi, and R. Kazman, "How does refactoring impact security when improving quality? A security-aware refactoring approach," *IEEE Trans. Softw. Eng.*, to be published, doi: 10.1109/TSE.2020.3005995.

[70] G. Bavota, B. De Carluccio, A. De Lucia, M. Di Penta, R. Oliveto, and O. Strollo, "When does a refactoring induce bugs? An empirical study," in *Proc. 12th Int. Work. Conf. Source Code Anal. Manipulation*, 2012, pp. 104–113.

[71] D. L. Parnas, "Software aging," in *Proc. 16th Int. Conf. Softw. Eng.*, 1994, pp. 279–287.

[72] S. Bartsch, "Practitioners' perspectives on security in agile development," in *Proc. 6th Int. Conf. Availability Rel. Secur.*, 2011, pp. 479–484.

[73] A. Poller, L. Kocksch, S. Türpe, F. A. Epp, and K. Kinder-Kurlanda, "Can security become a routine? A study of organizational change in an agile software development group," in *Proc. ACM Conf. Comput. Supported Cooperative Work Soc. Comput.*, 2017, pp. 2489–2503.

[74] D. Di Nucci, F. Palomba, G. De Rosa, G. Bavota, R. Oliveto, and A. De Lucia, "A developer centered bug prediction model," *IEEE Trans. Softw. Eng.*, vol. 44, no. 1, pp. 5–24, Jan. 2018.

[75] A. Zeller, *Why Programs Fail: A Guide to Systematic Debugging*. San Mateo, CA, USA: Morgan Kaufmann, 2005.

[76] T. W. Thomas, M. Tabassum, B. Chu, and H. Lipford, "Security during application development: An application security expert perspective," in *Proc. CHI Conf. Hum. Factors Comput. Syst.*, 2018, pp. 1–12.

[77] T. Scholte, D. Balzarotti, and E. Kirda, "Have things changed now? an empirical study on input validation vulnerabilities in web applications," *Comput. Secur.*, vol. 31, no. 3, pp. 344–356, 2012. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404811001684

[78] A. Avancini and M. Ceccato, "Security testing of web applications: A search-based approach for cross-site scripting vulnerabilities," in *Proc. IEEE 11th Int. Work. Conf. Source Code Anal. Manipulation*, 2011, pp. 85–94.

[79] M. Ceccato, C. D. Nguyen, D. Appelt, and L. C. Briand, "Sofia: An automated security oracle for black-box testing of SQL-injection vulnerabilities," in *Proc. 31st IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2016, pp. 167–177.

[80] M. Liu, K. Li, and T. Chen, "Security testing of web applications: A search-based approach for detecting sql injection vulnerabilities," in *Proc. Genetic Evol. Comput. Conf. Companion*, 2019, pp. 417–418.

[81] B. Aloraini, M. Nagappan, D. M. German, S. Hayashi, and Y. Higo, "An empirical study of security warnings from static application security testing tools," *J. Syst. Softw.*, vol. 158, 2019, Art. no. 110427. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121219302018
[82] Cppcheck, 2022. [Online]. Available: http://cppcheck.sourceforge.net/
[83] Flawfinder, 2022. [Online]. Available: https://dwheeler.com/flawfinder/
[84] Spotbugs, 2022. [Online]. Available: https://spotbugs.github.io/
[85] H. Borges, A. Hora, and M. T. Valente, "Understanding the factors that impact the popularity of GitHub repositories," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2016, pp. 334–344.
[86] F. Pecorelli, F. Palomba, F. Khomh, and A. De Lucia, "Developer-driven code smell prioritization," in *Proc. 17th Int. Conf. Mining Softw. Repositories*, 2020, pp. 220–231.
[87] G. Fraser and A. Arcuri, "Whole test suite generation," *Trans. Softw. Eng.*, vol. 39, no. 2, pp. 276–291, Feb. 2013.
[88] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, and A. De Lucia, "Automatic test case generation: What if test code quality matters?" in *Proc. 25th Int. Symp. Softw. Testing Anal.*, 2016, pp. 130–141.
[89] L. Pascarella, D. Spadini, F. Palomba, M. Bruntink, and A. Bacchelli, "Information needs in contemporary code review," *Hum.-Comput. Interact.*, vol. 2, 2018, Art. no. 135.
[90] G. Catolino, F. Palomba, A. De Lucia, F. Ferrucci, and A. Zaidman, "Enhancing change prediction models using developer-related factors," *J. Syst. Softw.*, vol. 143, pp. 14–28, 2018.
[91] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, "Understanding and detecting real-world performance bugs," *ACM SIGPLAN Notices*, vol. 47, no. 6, pp. 77–88, 2012.
[92] G. Catolino, F. Palomba, D. A. Tamburri, A. Serebrenik, and F. Ferrucci, "Gender diversity and women in software teams: How do they affect community smells?," in *Proc. 41st Int. Conf. Softw. Eng. Soc.*, 2019, pp. 11–20.
[93] J. Reason and C. U. Press, *Human Error*. Cambridge, U.K.: Cambridge Univ. Press, 1990. [Online]. Available: https://books.google.it/books?id=WJL8NZc8lZ8C
[94] B. Nagaria and T. Hall, "How software developers mitigate their errors when developing code," *IEEE Trans. Softw. Eng.*, to be published, doi: 10.1109/TSE.2020.3040554.
[95] H. Perl *et al.*, "VCCFinder: Finding potential vulnerabilities in open-source projects to assist code audits," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur.*, 2015, pp. 426–437.
[96] L. Yang, X. Li, and Y. Yu, "VulDigger: A just-in-time and cost-aware tool for digging vulnerability-contributing changes," in *Proc. IEEE Global Commun. Conf.*, 2017, pp. 1–7.
[97] E. Sahal and A. Tosun, "Identifying bug-inducing changes for code additions," in *Proc. 12th ACM/IEEE Int. Symp. Empir. Softw. Eng. Meas.*, 2018, pp. 1–2.
[98] 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," *Inf. Softw. Technol.*, vol. 57, pp. 369–377, 2015.
[99] A. Hindle, D. M. German, M. Godfrey, and R. Holt, "Automatic classication of large changes into maintenance categories," in *Proc. Int. Conf. Program Comprehension*, 2009, pp. 30–39.
[100] E. Murphy-Hill and A. P. Black, "Refactoring tools: Fitness for purpose," *IEEE Softw.*, vol. 25, no. 5, pp. 38–44, Sep./Oct. 2008.
[101] E. Murphy-Hill, C. Parnin, and A. Black, "How we refactor, and how we know it," *IEEE Trans. Softw. Eng.*, vol. 38, no. 1, pp. 5–18, Jan./Feb. 2012.
[102] G. Rodriguezperez, M. Nagappan, and G. Robles, "Watch out for extrinsic bugs! a case study of their impact in just-in-time bug prediction models on the openstack project," *IEEE Trans. Softw. Eng.*, to be published, doi: 10.1109/TSE.2020.3021380.
[103] M. Tufano *et al.*, "There and back again: Can you compile that snapshot?," *J. Softw. Evol. Process*, vol. 29, no. 4, 2017, Art. no. e1838.
[104] R. Henderson, M. Jones, and J. Stare, "Accuracy of point predictions in survival analysis," *Statist. Med.*, vol. 20, no. 20, pp. 3083–3096, 2001.
[105] P. Heagerty and Y. Zheng, "Survival model predictive accuracy and ROC curves," *Biometrics*, vol. 61, no. 1, pp. 92–105, 2005.
[106] R. Chillarege *et al.*, "Orthogonal defect classification-a concept for in-process measurements," *IEEE Trans. Softw. Eng.*, vol. 18, no. 11, pp. 943–956, Nov. 1992.
[107] G. Canfora, A. Di Sorbo, S. Forootani, A. Pirozzi, and C. A. Visaggio, "Investigating the vulnerability fixing process in OSS projects: Peculiarities and challenges," *Comput. Secur.*, vol. 99, 2020, Art. no. 102067.
[108] J. Smith, B. Johnson, E. Murphy-Hill, B. Chu, and H. Richter, "How developers diagnose potential security vulnerabilities with a static analysis tool," *IEEE Trans. Softw. Eng.*, vol. 45, no. 9, pp. 877–897, Sep. 2019.

Emanuele Iannone is currently working toward the PhD degree with the University of Salerno, Fisciano, Italy. His main research focuses on the analysis of software vulnerabilities, particularly in the broader context of software security testing. His research interests also include the development of novel tools and techniques of mining software repositories and search-based software testing. He also served as a reviewer for the main international conferences and journals of the software engineering field.

Roberta Guadagni received the MSc degree from the University of Salerno, Fisciano, Italy, in 2019. She is currently a machine learning engineer with Senseledge, where she develops machine learning and MLOps solutions for real-world problems. Formerly, she was a data scientist with Whitehall Reply.

Filomena Ferrucci is a professor of software engineering and software project management at the University of Salerno, Italy. Her main research interests include software metrics, effort estimation, empirical software engineering, search-based software engineering, and human-computer interaction. She coauthored more than 150 publications in international journals, book chapters, and conferences. She took part in the program committees of many international conferences in the field of software engineering.

Andrea De Lucia (Senior Member, IEEE) is a professor of software engineering at the University of Salerno, Italy. His research interests include software maintenance and testing, reverse engineering and reengineering, source code analysis, and traceability management. He has published more than 250 papers in international journals, books, and conference proceedings. He is co-editor in chief of Science of Computer Programming (Elsevier) and serves on the editorial board of *Empirical Software Engineering* (Springer) and *Journal of Software Evolution and Process* (Wiley). He was member-at-large of the executive committee of the IEEE Technical Council on Software Engineering.

Fabio Palomba received the European PhD degree in management & information technology in 2017. He is an Assistant professor at the University of Salerno, Italy. His research interests include software maintenance and evolution, software testing, empirical software engineering, and source code quality. He serves and has served as an organizing and program committee member of various international conferences and as editorial board member of flagship software engineering journals.