

An Exploratory Study on Self-Fixed Software Vulnerabilities in OSS Projects

Sara Forootani*, Andrea Di Sorbo*, Corrado A. Visaggio*

*University of Sannio, Italy

{forootani, disorbo, visaggio}@unisannio.it

Abstract—Remediation to a software vulnerability can be accomplished either by the developer who introduced it or by a different one. In this context, we refer to a *self-fixed vulnerability* when the fixing is carried out by the developer who introduced it. Previous research demonstrated that a developer who introduces a bug is also the best candidate to fix it. However, as vulnerabilities conceptually differ from non-security bugs and specific skills and knowledge are required for solving them, it is unclear if the previous finding also applies to vulnerabilities or specific vulnerability types. To fill this gap, in this paper, we investigate the diffusion of self-fixed vulnerabilities within software projects, the types of vulnerabilities that are more prone to self-fixing, and the time required to solve self-fixed vulnerabilities compared to non-self-fixed ones. Specifically, we analyzed 1,752 commits related to C and PHP open-source projects aimed at fixing (or self-fixing) vulnerabilities spanning 17 different types of software weaknesses. The results of our study show that 20.55% of the considered vulnerabilities in C projects and 36.46% of the considered vulnerabilities in PHP projects are self-fixed. In addition, the average remediation time of self-fixed vulnerabilities is generally shorter than non-self-fixed ones. In particular, in C projects, self-fixed integer overflow vulnerabilities are patched about 5 times shorter than non-self-fixed ones, while vulnerabilities related to improper calculation or conversion of numbers are generally fixed faster by other developers. Similarly, in PHP projects, CSRF vulnerabilities tend to be patched in a shorter time when they are self-fixed, while unauthorized access vulnerabilities are likely repaired faster by other developers. Our results can help both researchers and practitioners identifying the best candidates to solve specific vulnerability bugs.

Index Terms—Software Vulnerabilities Remediation, Software Maintenance, Empirical Study

I. INTRODUCTION

Motivation. Organizations pose little attention to the remediation of vulnerabilities, and this lack of attention could have negative effects also on the vulnerability discovery efforts [2]. In particular, the density of security vulnerabilities in a software project is related to the distribution of the different types of interaction patterns that arise among developers during security-related activities [39]. Thus, considering vulnerability remediation management is important, especially from the perspective of developers. More specifically, a vulnerability may be fixed by either the same developer who introduced it in the first place, i.e., *self-fixed* vulnerability, or by other developers. While previous research on vulnerability management mainly focused on investigating vulnerability discovery and remediation strategies from the source code perspective (e.g., patching actions) [4], [31], [40], [30], [15], only a few studies looked at the problem from the developers' angle [5],

[39]. Indeed, very little is known about the extent to which the self-fixing phenomenon is widespread, and investigating this phenomenon can help identify good and bad practices in vulnerability remediation, especially when comparing the vulnerabilities that are self-fixed with vulnerabilities that are not. In particular, previous research [27], [5] demonstrated that it could be difficult to find the best candidates to fix security bugs due to the scarcity of developers with security-related skills. In addition, security bugs are usually assigned more developers and are more reopened than other types of bugs [41]. One common reason for tossing is an incorrect assignment to a developer who is not familiar with the defective source or may not have the expertise to fix the bug [13]. This not only wastes the developers' time but also reduces the efficiency of the bug assignment process [1]. Indeed, previous research showed that developers fix their own introduced bugs more quickly, as they are more familiar with the implemented code, resulting in higher efficiency of the bug fixing process [38]. Such results have shown to be applicable for bug assignment recommendations systems [13], [21], [11], [36]. However, as vulnerabilities conceptually differ from non-security bugs and specific skills and knowledge are required to fix them [24], [5], it is unclear if previous findings also apply to vulnerabilities or specific vulnerability types. On the one hand, investigating the vulnerabilities that are more likely and more quickly self-fixed could help in better understanding the vulnerability types whose fixes are best suited to be assigned to the developers who introduced them and, therefore, obtain more balanced vulnerability patching workloads. On the other hand, a similar investigation also helps in better comprehending the vulnerability types whose fixes more likely need the contribution of security experts. However, there is not yet evidence about the extent to which this phenomenon occurs in practice and the types of vulnerabilities that are mainly self-fixed. Such an investigation contributes to shedding light on the process of vulnerability remediation and point out recommendations for improving it. It also allows a better understanding of whether the self-fixing phenomenon may have positive effects on reducing the exposure time of a vulnerability.

Aim and scope. To the best of the authors' knowledge, this is the first study that quantitatively and qualitatively investigates the self-fixing phenomenon of software vulnerabilities. In particular, our main purpose is to investigate the distribution of self-fixed vulnerabilities in software projects, the types of vulnerabilities that are more prone to self-fixing, and how long

self-fixed vulnerabilities take to be corrected, compared to non-self-fixed ones. To carry out the study, we examined 1,752 commits pertaining to 138 and 92 projects written in C and PHP programming languages respectively. In particular, the extracted vulnerabilities cover 17 different types of software weaknesses. To identify the vulnerability fixing commits, we analyzed the vulnerability reports using python scripts written by the authors. In addition, the GitHub blame technique¹ has been leveraged to detect the commits introducing the corresponding vulnerabilities.

The Results indicate that the phenomenon of self-fixed vulnerabilities is not that common among developers. This might be due to a lack of knowledge in writing secure code [5] and not applying security tests accordingly. Furthermore, for both investigated languages (C and PHP) we found that the average survival time of self-fixed vulnerabilities is generally shorter than non-self-fixed ones. In particular, in C projects, integer overflow vulnerabilities are more likely to be self-fixed. In addition, the remediation time of self-fixed integer overflow vulnerabilities is about 5 times shorter than non-self-fixed ones, while the vulnerabilities related to improper calculation or conversion of numbers are fixed more and faster by other developers. Instead, in PHP projects, developers more often self-fix SQL injection and CSRF vulnerabilities. However, vulnerabilities depending on improper access restrictions to resources are repaired more (82%) and faster (one time shorter) by other developers.

Contribution. Our findings can benefit both researchers and practitioners. The self-fixing rates of the different vulnerability types could be useful to develop recommendation systems supporting vulnerability remediation tasks. For example, such tools can benefit from our findings by assigning the remediation of specific vulnerabilities to the developers who likely introduced them or to a set of appropriate developers who frequently have been modifying vulnerable code portions.

Moreover, our findings can help further explore the role of developers and team leaders in devising strategies for fixing vulnerabilities. We also make the dataset used in our study available in a replication package. The dataset comprises the details of the vulnerability fixing commits at the hunk level, and it also encompasses the information related to the commits that introduced the vulnerabilities.

Paper structure. Section 2 describes the study design, while Section 3 illustrates the results of our study. Section 4 discusses the possible implications, and Section 5 deals with the threats to the study's validity. After the discussion of related work in Section 6, Section 7 concludes the paper.

II. STUDY DESIGN

The design of the empirical study is based on the guidelines derived from Runeson *et al.* [29], while the Linear Analytic Structure [29] is adopted to report it.

¹The Git blame is used to investigate the content of a file line by line and see when and where each line was last modified and who the author of the modifications was.

A. Objectives and Research Questions

The *goal* of our study, established according to the Goal-Question Metrics (GQM) approach [37], is to analyze software systems for the *purpose* of investigating the vulnerability self-fixing phenomenon *with respect to* its diffusion in the different languages and vulnerability types, and its relations with the resolution times, *from the point of view* of developers in the *context* of open-source software projects.

Based on the above goal, we define the three research questions reported below:

- **RQ₁:** *To which extent are software vulnerabilities self-fixed in the studied open source projects?* Ideally, a developer introducing a vulnerability in the system is also expected to be the best candidate to fix it. Indeed, developers are more aware of their code, and, consequently, possible vulnerability root causes or potential remediation strategies could be more effectively identified. Thus, this research question aims to investigate the extent to which developers take care of their own introduced vulnerabilities.
- **RQ₂:** *Do different vulnerability types show different self-fixing rates?* Different types of vulnerabilities require different amounts of effort to be fixed [16]. With regards to this research question, we investigate which vulnerability types are more likely to be fixed by the developers who introduced them. In particular, we provide insights on how developers consider their own introduced vulnerabilities and prioritize their remediation. For instance, some types of vulnerabilities may be associated with higher risks and, thus, they may be considered as important to fix by the developers who introduced them. Some other vulnerability types may be considered as easy to self-fix. Besides, identifying the vulnerability types mainly fixed by others may help to understand the vulnerabilities that more likely require the intervention of security experts.
- **RQ₃:** *Are self-fixed software vulnerabilities resolved faster?* This research question aims to understand whether self-fixed vulnerabilities are fixed faster than those fixed by developers who did not introduce them. Investigating the time to resolve self-fixed vulnerabilities helps better comprehend how developers prioritize vulnerability fixing activities and how the exposure time is consequently affected. Since developers tend to be more familiar with their own code, one would expect that developers can address the vulnerabilities introduced by themselves in a shorter time.

B. Context Selection

For the purpose of this study, we selected C and PHP projects.

Beyond being among the top 10 most used programming languages among developers², C and PHP are the languages with the highest numbers of vulnerabilities reported in the

²<https://www.tiobe.com/tiobe-index/>

last 10 years, 47% and 17% of all reported vulnerabilities, respectively³.

In addition, while C is a procedural typed general-purpose programming language, PHP is a highly flexible language, especially suited for web development, presenting a set of dynamic features that could affect software quality [28]. We believe that the structural differences between the two languages and the different purposes for which they are usually adopted allow us to observe a wider spectrum of vulnerability types and developer self-fixing behaviors as well.

Furthermore, to extract information on security patches of C and PHP projects, we mined data from the National Vulnerability Database (NVD)⁴. Although there exist several vulnerability databases (e.g., *securityfocus.com*, *IBM's X-Force Threat Intelligence*, and *securitytracker.com*), we chose to extract data from NVD, because it is: (i) detailed, containing extensive documentation of vulnerabilities (notably external references to source code repositories), (ii) public, free, and easily accessible in JSON and XML format, allowing for reproducibility and follow-up studies, (iii) extensive, as the NVD aims to catalog all publicly disclosed vulnerabilities across numerous software packages, and (iv) manually vetted and curated, which increases the accuracy of data.

C. Data Collection

In this section, we describe the process of data gathering. In particular, our data collection approach comprises two main phases: (i) detection of vulnerability fixing commits (detailed in Section II-C1), and (ii) detection of vulnerability-introducing commits (detailed in Section II-C2).

1) *Detection of vulnerability fixing commits/files*: We start our data collection from the NVD. In the following we describe the main steps followed to first collect vulnerability patches information:

- a. **Mining NVD Data.** Stemming from the NVD database, we collected in a JSON file the information about Common Vulnerabilities and Exposures (CVE) entries corresponding to a period ranging from 2002 up to 2020⁵.
- b. **Extracting URLs.** For each CVE in the aforementioned JSON file, we reverse-engineered the URLs in it referenced as external resources. We leveraged such URLs to obtain details about (i) the online source code repository type (e.g., git, svn, etc.), (ii) the project name, (iii) the main programming language, and (iv) the id/hash of the vulnerability fixing commit. Previous research [19] demonstrated that the links referenced by NVD reports pointing to specific commits reflect, in almost all cases, fixes for the corresponding vulnerabilities.
- c. **URLs Filtering.** Since we were interested in analyzing patches applied to software projects written in the C and PHP programming languages, we filtered out the URLs pointing to repositories not containing any *C* and *PHP* file.

In our case, the projects of interest were hosted on Git⁶ version control systems.

- d. **Vulnerability Patches.** For each identified patch we reported a triplet specifying the vulnerability identifier (i.e., CVE reference), the repository URL, and the commit id. For each entry in this collection, we used the gathered information to (i) identify and clone the referenced repository, and (ii) access to the code transformations applied by developers for patching the vulnerability identified by the CVE reference. We afterward, recorded the project name, commit hashcode, and developer responsible for the commit, including the developer's name and email information.
- e. **Security Weakness Information.** We leveraged the CVE references collected in the previous step to scrape the CVE Details website⁷ and assign to each CVE entry in our collection the corresponding Common Weakness Enumerations (CWE) id. More specifically, the CWE category provides information related to the root cause of each vulnerability. Thus, the CWE id has been used in turn to gain weakness category-specific information by crawling the CWE website⁸. At the end of this process, we were able to collect vulnerabilities belonging to 52 and 32 different weakness categories (i.e., CWEs) pertaining to C and PHP projects, respectively, and in total, we found 65 unique CWEs. However, we decided to only consider vulnerabilities belonging to the top 25 most dangerous software weaknesses⁹. Thus, out of 65 different CWEs, we only considered the vulnerabilities belonging to 12 CWEs mentioned among the most dangerous ones as reported in Table I. To guarantee higher variability in our data collection, we decided to also include in our analysis the 5 CWEs for which we had collected the higher numbers of vulnerabilities (these CWEs are highlighted with a super-scripted asterisk * in Table I).

```

1 diff --git a/fs/proc/proc_sysctl.c b/fs/proc/
  proc_sysctl.c
2 index d653907275419..7325baa8f9d47 100644
3 --- a/fs/proc/proc_sysctl.c
4 +++ b/fs/proc/proc_sysctl.c
5 @@ -1626,9 +1626,11 @@ static void
   drop_sysctl_table(struct ctl_table_header
   *header)
6     if (--header->nreg)
7         return;
8
9     - if (parent)
10    + if (parent) {
11        put_links(header);
12    - start_unregistering(header);
13    + }
14    +
15    if (!--header->count)
16        kfree_rcu(header, rcu);

```

Listing 1. File diff example

³<https://www.whitesourcesoftware.com/most-secure-programming-languages/>

⁴<https://nvd.nist.gov/>

⁵https://nvd.nist.gov/vuln/data-feeds#JSON_FEED

⁶<https://git-scm.com>

⁷<https://www.cvedetails.com>

⁸<https://cwe.mitre.org>

⁹https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html

f. **Patch File Separation.** A vulnerability is usually patched through code modifications. Such modifications are applied in specific commits, and a commit could affect different files. In each of the affected files, code modifications are applied at the line level. Thus, a patched file may involve multiple code modifications. In particular, in the context of a commit and a specific affected file, the number of unmodified rows jointly with the code modification section is called a hunk. For example, the file diff in Listing 1 shows one hunk: each hunk is prefixed by “@@” special characters and source code lines removed and/or added are anchored between each pair of “@@”. For each vulnerability fixing commit, we collected the involved files in each commit, the affected hunks in each file, and the lines deleted within each hunk. At the end of this step, we identified 3085 fixed hunks related to C projects and 2046 fixed hunks related to PHP projects, as reported in Table I.

2) *Detection of vulnerability-introducing commits:* Stemming from the collected vulnerability fixing commits, we further identified the related vulnerability-introducing commits by using the blame technique provided by the Version Control System (VCS), e.g., git. Similar to previous studies [26], [34], [6], [14] we assume that the deleted lines in a vulnerability fixing commit are related to the root cause of the vulnerability and, thus, they should be considered as faulty lines. In the context of our study, we consider a vulnerability-introducing commit as *the commit that introduces faulty lines for the first time*. Therefore, by using the blame technique on the faulty lines gathered from former steps, we got access to the commits where those lines were introduced for the first time. In particular, we identified vulnerability-inducing commits at the hunk level as (i) each faulty line in each hunk of a patch file may raise from different commits and (ii) the chosen strategy may affect the survival time and the type of fixing (self or non-self fixing) and (iii) accordingly the accuracy of the approach. For this reason, we categorized vulnerability hunks as (i) hunks for which we have all faulty lines that are raised by a single introduction commit, and (ii) hunks that contain faulty lines induced by multiple commits. Based on the SZZ algorithm [17], [18], for hunks that contain faulty lines induced by multiple commits, the minimum survival time is considered as the time a vulnerability is introduced (i.e., the most recent commit identified), while for hunks where we have all faulty lines that are raised by a single introduction commit the maximum survival time is considered (i.e., the least recent commit identified). Moreover, a manual analysis performed on some vulnerabilities confirmed the validity of our experimental design to assess the survival time of vulnerabilities.

In this step, we also recorded metadata of these commits such as commit id, commit date, commit message, author’s name and email for later computation.

D. Data Analysis

To conduct our analysis, we defined metrics similar to the study carried out by Tan *et al.* [35], such as (a) the count of self-fixed and non-self-fixed vulnerabilities for each CWE, and

(b) the remediation time for each self-fixed and non-self-fixed vulnerability.

To answer RQ₁, we compared the number of vulnerabilities that were self-fixed with the non-self-fixed ones (i.e., fixed by a developer different from the one who introduced the faulty lines) for each programming language. In particular, we computed the self-fixing rate for both C and PHP projects, i.e., the percentage of vulnerabilities that are fixed by the same developers who introduced them, which is measured as:

$$\text{Self-fixing rate}(pl) = \frac{\text{num_of_self_fixed}_{pl}}{\text{num_of_fixed}_{pl}}$$

where, for a programming language pl , $\text{num_of_self_fixed}_{pl}$ and num_of_fixed_{pl} are the number of self-fixed vulnerabilities and the total number of fixed vulnerabilities, respectively, for all the projects written in the pl programming language.

Similarly, to answer RQ₂, for each CWE, we computed the self-fixing rate to investigate whether vulnerabilities belonging to specific CWE categories are self-fixed more often than others. In particular, we computed two different metrics. To investigate whether some CWEs that are common between C and PHP projects are self-fixed more often than others, we computed:

Total self-fixing rate: Given a specific CWE w , it computes the percentage of vulnerabilities belonging to w and affecting either C or PHP projects that have been self-fixed.

$$\text{Total Self-fixing rate}(w) = \frac{\sum_{pl=1}^n \text{num_of_self_fixed}_{pl,w}}{\sum_{pl=1}^n \text{num_of_fixed}_{pl,w}}$$

Where n is the number of programming languages (i.e., 2), $\text{num_of_fixed}_{pl,w}$ is the number of fixed vulnerabilities belonging to w for the projects written in the pl programming language and $\text{num_of_self_fixed}_{pl,w}$ is the number of self-fixed vulnerabilities belonging to w for the projects written in the pl programming language.

However, the total self-fixing rate for common CWEs across different languages may be biased, as imbalanced numbers of vulnerabilities are considered for each language. In particular, if a higher number of vulnerabilities is considered for one of the languages, then the value of the total self-fixing rate would approach the self-fixing rate of that language. To mitigate this issue, we made use of another metric called the average self-fixing rate.

Average self-fixing rate: Given a CWE w , it represents the average of the self-fixing rates for the vulnerabilities belonging to w obtained in each programming language pl .

$$\text{Average self-fixing rate}(w) = \frac{1}{n} \sum_{pl=1}^n \frac{\text{num_of_self_fixed}_{pl,w}}{\text{num_of_fixed}_{pl,w}}$$

TABLE I
DISTRIBUTION OF THE SELECTED CWES OVER PROGRAMMING LANGUAGES

Software Weakness	C project	PHP project	Definition
cwe-200	147	128	Exposure of Sensitive Information
cwe-264*	186	163	Permissions, Privileges, and Access Control
cwe-284*	173	202	Improper Access Control
cwe-20	325	70	Improper Input Validation
cwe-89	8	616	Sql injection
cwe-22	3	147	Improper Limitation of Pathname to a Restricted Directory
cwe-787	159	1	Out-of-bounds Write
cwe-476	148	-	NULL Pointer Dereference
cwe-119	724	-	improper restriction of operations within the bounds of a memory buffer
cwe-125	468	-	Out-of-bounds Read
cwe-189*	117	-	Numeric Errors
cwe-399*	171	-	Resource Management Errors
cwe-190	154	-	Integer Overflow or Wraparound
cwe-416	209	-	Use After Free
cwe-362*	93	-	Race Condition
cwe-79	-	583	Improper Neutralization of Input During Web Page Generation
cwe-352	-	136	Cross-Site Request Forgery (CSRF)
Total-Hunk	3085	2046	

where n , $num_of_fixed_{pl,w}$ and $num_of_self_fixed_{pl,w}$ are explained as above. This formula represents the average percentage of self-fixed issues, which is not affected by the different number of issues in the different languages.

To answer RQ₃, we computed and analyzed the survival time of each vulnerability. This metric measures the number of days elapsed between the introduction of a vulnerability and its resolution. In particular, we computed the average survival time for self-fixed and non-self-fixed vulnerabilities for each considered CWE and programming language and compared the obtained values.

Finally, we applied the Mann-Whitney non-parametric test (with p-value fixed at 0.05) to check whether statistically differences could be observed between the resolution times of self-and non-self-fixed vulnerabilities belonging to each specific CWE. Specifically, we used a non-parametric statistical test as the variables of interest are not well-modeled by normal distributions (as verified through the Shapiro-Wilk test [33]). Besides, to measure the magnitude of the emerging differences, Cliff's $|d|$ effect size measure [9] was used. As recommended by the guidelines in [9], we interpreted the effect size as *small* for $|d| < 0.33$, *medium* for $0.33 \leq |d| < 0.474$, and *large* for $|d| \geq 0.474$.

Replication package We make available in our replication package¹⁰ the dataset used in our study.

¹⁰https://www.dropbox.com/sh/9bl8jcz4nxa9pxs/AACISRbp19Nm33V_MopYGY2ya?dl=0

TABLE II
DETAILS OF THE SELECTED CWES OVER C AND PHP PROGRAMMING LANGUAGES

	C	PHP
Commits	1395	357
Files	1666	1058
Hunks	3085	2046
SLOC	18074	10591
Self-fixing Rate	20.55%	36.46%

III. RESULTS

A. *To which extent are software vulnerabilities self-fixed in the studied open source projects?*

To understand the extent to which the self-fixing phenomenon occurs in vulnerability remediation, in Table II we provide the following details about the two selected C and PHP programming languages: the total number of commits, files, hunks, deleted lines of code (SLOC), and the rate of self-fixed hunks.

In particular, for C projects, as reported in Table II, out of 3085 fixed hunks only 20.55% are fixed by the same developers who introduced them. Similarly, for PHP projects, out of 2046 fixed hunks, 36.46% of them have been fixed by the author. These results indicate that, in both C and PHP projects, vulnerabilities are mostly resolved by developers different from the ones who likely introduced them. Thus, the phenomenon of vulnerability self-fixing only occurs in less than 40% of cases. This could be due to the developers' lack of knowledge or insufficient experience in secure coding practices [2], [22], [5].

RQ₁ Summary: For both *C* and *PHP* projects, the majority of software vulnerabilities are resolved by developers different from who introduced them.

B. Do different vulnerability types show different self-fixing rates?

Tables III and IV report the numbers of self-fixed hunks and non-self-fixed hunks along with their self-fixing rates in descending order for the different CWE categories among *C* and *PHP* projects, respectively. In particular, for *C* projects, CWE-190 has the highest self-fixing rate (55%), followed by CWE-89 (37%). Therefore, developers are more likely to self-fix integer overflow vulnerabilities (e.g., when an arithmetic operation exceeds the maximum size of the integer type used to store its result) compared to other types of vulnerabilities. This result could be due to the fact that localizing integer overflow vulnerabilities in the *C* language is easy, as this language does not support automated bounds checking. In addition, it is possible to handle such vulnerabilities through libraries or frameworks (e.g., IntegerLib¹¹). In contrast, CWE-416 has the lowest self-fixing rate (9%) among the 15 CWEs considered for this language. This result indicates that the vulnerabilities arising by referencing previously-freed memory receive less attention or lower priority from the developers who introduce them, as they might find it difficult to fix.

Instead, for *PHP* projects, as shown in Table IV, the highest self-fixing rate is achieved by the CWE-89 category (i.e., 57%), while the lowest self-fixing rate is obtained for CWE-22 (i.e., 10%).

To investigate the self-fixing rate of common CWEs between the two programming languages, *C* and *PHP*, Table V shows for each common CWE: the average self-fixing rate (i.e., based on the self-fixing rates per language), and the total self-fixing rate (i.e., based on all fixed vulnerabilities). In Table V, CWEs are sorted in decreasing order of average self-fixing rates. This can help to investigate the developers' vulnerability remediation habits in the different programming languages. Hence, we observe that the values of average self-fixing rate (i.e., ASFR) and total self-fixing rate (i.e., TSFR) for CWE-284 are equal (14%), likely meaning that developers have the same attitude when dealing with vulnerabilities caused by improper access restrictions to the resources, despite the programming language. In addition, CWE-89 exhibits the highest value of average self-fixing and total self-fixing rate, suggesting that SQL injection vulnerabilities are more likely fixed by the developers who authored them regardless of the language. This could be due to the fact that SQL injection is one of the most common vulnerabilities¹², and there are plenty of resources that can be referred to fix such vulnerabilities. SQL injection has great incidence also in our dataset. However, to have a high incidence does not mean that the developers are able to deal with the problem. For example, among *C* projects, CWE-119 is among the ones occurring most often, but its self-fixing rate is not amongst the highest ones.

¹¹<https://cwe.mitre.org/data/definitions/190.html>

¹²<https://www.veracode.com/security/owasp-top-10>

TABLE III
THE NUMBER OF *self-fixed-hunk (SF-h)*, *non-self-fixed-hunk (NSF-h)*, AND OF *Self-fixing rates (SFR)* FOR INDIVIDUAL CWEs AMONG *C* PROJECTS

CWE	#SF-h	#NSF-h	SFR
cwe-190	84	70	0.55
cwe-89	3	5	0.37
cwe-22	1	2	0.33
cwe-476	46	102	0.31
cwe-20	96	229	0.30
cwe-787	45	114	0.28
cwe-119	156	568	0.21
cwe-189	20	97	0.17
cwe-284	27	146	0.15
cwe-264	26	160	0.14
cwe-200	21	126	0.14
cwe-125	58	410	0.12
cwe-399	21	150	0.12
cwe-362	10	83	0.10
cwe-416	20	189	0.09
Average			0.22
Median			0.17

TABLE IV
THE NUMBER OF *self-fixed-hunk (SF-h)*, *non-self-fixed-hunk (NSF-h)*, AND OF *self-fixing rates (SFR)* FOR INDIVIDUAL CWEs AMONG *PHP* PROJECTS

CWE	#SF-h	#NSF-h	SFR
cwe-89	357	259	0.57
cwe-352	75	61	0.55
cwe-200	62	66	0.48
cwe-264	74	89	0.45
cwe-79	123	460	0.21
cwe-20	11	59	0.15
cwe-284	29	173	0.14
cwe-22	15	132	0.10
Average			0.335
Median			0.332

RQ₂ Summary: While in *C* projects developers are more likely to self-fix integer overflow vulnerabilities, in *PHP* projects, SQL injection vulnerabilities are the ones more likely fixed by the developers who introduced them. Vulnerabilities caused by improper access restrictions to resources receive equal attention from developers who introduced them in both languages.

TABLE V
Average *self-fixed rates (ASFR)* AND Total *Self-fixing rates (TSFR)* FOR COMMON CWEs BETWEEN *C* AND *PHP* PROJECTS

CWE	ASFR	TSFR
cwe-89	0.47	0.57
cwe-200	0.31	0.30
cwe-264	0.29	0.28
cwe-20	0.22	0.27
cwe-284	0.14	0.14
cwe-22	0.21	0.10

C. Are self-fixed software vulnerabilities resolved faster?

In Tables VI and VII, we show the average survival time of self-fixed vulnerabilities (AT-SF) and non-self-fixed vulnerabilities (AT-NSF) in a descending order based on the ratio of AT-NSF to AT-SF for each considered CWE concerning C projects (see Table VI) and PHP projects (see Table VII). For **C projects**, the average survival time of self-fixed vulnerabilities is shorter than the non-self-fixed ones, for most of the considered CWEs (13 out of 15). For example, looking at Table VI, we can observe that the self-fixed vulnerabilities belonging to CWE-190 are fixed more than 5 times faster than non-self-fixed ones, on average. Similar results are achieved for almost all types of CWEs (with the noticeably exception of CWE-284) in **PHP projects** (see Table VII). Indeed, self-fixed vulnerabilities belonging to CWE-352 are fixed more than 9 times faster than non-self-fixed ones, on average. These results suggest that for most of the considered CWEs, the vulnerability average survival time is reduced when the vulnerability is resolved by the same developer introducing it, regardless of the programming language. This could depend on the fact that the familiarity with their own introduced code makes it easier for developers to identify and repair the vulnerability.

We further investigated whether statistically significant differences in the resolution times of self and non-self fixed vulnerabilities can be observed for each investigated CWE and programming language. More specifically, for C projects, as shown in Table VIII, we obtain statistically significant differences with *large* effect size for three CWE categories: CWE-190 ($|d| = 0.615$), CWE-20 ($|d| = 0.675$) and CWE-264 ($|d| = 0.506$). For the other CWEs reported in Table VIII, the effect sizes are *small*. For PHP projects, as shown in Table IX, we observe a *large* effect size ($|d| = 0.661$) between the resolution times of self- and non-self- fixed vulnerabilities belonging to CWE-352, while *medium* effect size is found when considering CWE-79 and *small* effect size is observed when looking at vulnerabilities belonging to CWE-264. It is worth noticing that, for simplicity, in Tables VIII and IX, we only reported CWEs for which we obtained a p-value < 0.05 .

To achieve further insights on the survival time of self-fixed vulnerabilities and non-self-fixed ones, for each CWE, we grouped the self-fixed and non-self-fixed vulnerabilities based on the different time spans required for their resolution. In particular, Figures 1 and 2 show the distributions of survival times of the self-fixed (right side) and the non-self-fixed (left side) vulnerabilities for both C and PHP projects, respectively. The percentages represent how often a vulnerability was fixed within a certain time span (e.g., one week, one month, one year, and more). The time spans are reported in different colors, ranging from *less than or one day* to *more than one year*. CWEs are sorted by the difference in the percentage between the two groups of non-self and self-fixed vulnerabilities that are resolved within the *less than or one day* time span (the pink parts in the middle of the figures) in descending order (the greater the difference, the lower the position in

the Figures). As shown in Figure 1, in **C projects**, for all the CWEs below to CWE-189, the percentages of self-fixed vulnerabilities resolved in the *less than or one day* time span (from CWE-125 to bottom in the Figure) are higher than those of non-self-fixed vulnerabilities.

For CWE-284, CWE-89, CWE-22 and CWE-190 we do not observe self-fixed vulnerabilities that are resolved within the *less than or one day* time span. However, while for CWE-284, CWE-89, and CWE-22 we do not even find non-self-fixed vulnerabilities that are resolved in this time interval, for CWE-190 we report a small percentage (i.e., 8.57%) of non-self-fixed vulnerabilities that are resolved in a time span of one day. In contrast, for CWE-416, CWE-399, and CWE-264, we only report self-fixed vulnerabilities that are resolved in *less than or one day*. Moreover, comparing the rest of the time spans, we see that **for each time range shorter than one year, the percentage of the self-fixed vulnerabilities is most of the times larger than non-self-fixed ones, while, for the time range longer than one year, it happens the opposite**. It is noteworthy that, for CWE-190 (71.42%), CWE-20 (67.70%), and CWE-264 (46.15%), the self-fixing of these kinds of vulnerabilities occurs within a period ranging from one month to one year. Finally, CWE-22 vulnerabilities are mainly repaired within more than one year, regardless of whether they are fixed by the same developer who introduced them or not.

Similarly, as shown in Figure 2, in **PHP projects**, for all the CWEs below to CWE-89 the percentages of self-fixed vulnerabilities resolved in the *less than or one day* time span are higher than those of non-self-fixed vulnerabilities, while, for the CWEs above CWE-284, we found the opposite, suggesting that vulnerabilities belonging to such CWEs and fixed by developers different from who introduced them are resolved in a shorter time.

It is noteworthy that, looking at CWE-22, about one-fourth (i.e., 26%) of the self-fixed vulnerabilities turn out to be fixed by the same developer who introduced them within a day. While most of the self-fixed vulnerabilities belonging to CWE-352 (57.33%) and CWE-79 (51.21%) are resolved within the time span ranging from one month and one year, for the other CWEs categories, self-fixed vulnerabilities are mostly resolved within a period of more than one year. It is worth noticing that we did not find CWE-787 instances that are self-fixed and all non-self-fixed vulnerabilities in this category remained unfixed for over a year.

Finally, as reported in Table X, we compute the average fixing time of self-fixed and non-self-fixed total vulnerabilities together with the ratio between the two values for each programming language. As listed in Table X, for both C and PHP, the self-fixed vulnerabilities are resolved around one-half faster on average than the time required to resolve non-self-fixed vulnerabilities.

For the majority of CWEs, both self-fixed and non-self-fixed vulnerabilities are more likely resolved within a period longer than one year. This could depend on the fact that vulnerabilities are often discovered late in the development

TABLE VI
AVERAGE SURVIVAL TIME OF SELF-FIXED (AT-SF) AND OF NON
SELF-FIXED (AT-NSF) FOR DIFFERENT CWEs OVER C LANGUAGES

CWE	AT-SF (days)	AT-NSF (days)	AT-NSF/AT-SF
cwe-190	380.16	2124.35	5.58
cwe-20	373.22	2079.95	5.57
cwe-22	453	2496	5.50
cwe-399	434.09	1551.34	3.57
cwe-476	522.52	1831.01	3.50
cwe-284	611.33	1578.51	2.58
cwe-264	460.76	1187.8	2.57
cwe-416	975.15	2074.12	2.12
cwe-362	924.15	1932.10	2.09
cwe-200	825.85	1445.68	1.75
cwe-125	1191.63	1753.84	1.47
cwe-119	1534.712	2069.95	1.34
cwe-787	1596.95	2097.91	1.31
cwe-189	1452.3	1417.78	0.97
cwe-89	3857.33	1400	0.36

TABLE VII
AVERAGE SURVIVAL TIME OF SELF-FIXED (AT-SF) AND OF NON
SELF-FIXED (AT-NSF) FOR DIFFERENT CWEs OVER PHP LANGUAGES

CWE	AT-SF (days)	AT-NSF (days)	AT-NSF/AT-SF
cwe-352	226.48	2129.37	9.40
cwe-22	134.26	801.58	5.97
cwe-79	384.75	1504.71	3.91
cwe-20	350.09	1244.37	3.55
cwe-200	653.74	1344.19	2.05
cwe-264	598.98	1011.29	1.68
cwe-89	1095.98	1525.35	1.39
cwe-284	2123.82	1635.99	0.77

process [24], increasing the related exposure time.

RQ₃ Summary: *Self-fixed vulnerabilities are usually resolved earlier compared to non-self-fixed ones. In C projects, self-fixed buffer overflow vulnerabilities are repaired about 5 times faster than non-self-fixed ones, on average. Instead, in PHP projects, self-fixed cross-site request forgery vulnerabilities are resolved about 9 times faster than non-self-fixed ones, on average. In general, the number of self-fixed vulnerabilities that are resolved in less than one year is higher than non-self-fixed vulnerabilities.*

IV. IMPLICATIONS

Our study sheds some more light on the phenomenon of self-fixed software vulnerabilities, investigating the types of vulnerabilities that are more self-fixed, and the remediation time of self-fixed vulnerabilities compared to non-self fixed ones. Specifically, both *researchers* and *practitioners* can benefit from our findings. In the following, we briefly summarize some possible practical implications of our findings for the research as well as the industrial community.

Practical implications for researchers. The self-fixing rates drawn from this study can be leveraged by researchers to

TABLE VIII
THE RESULTS OF P-VALUE TOGETHER WITH CLIFF'S $|d|$ EFFECT SIZE ON
SURVIVAL TIME FOR SELF VS NON-SELF-FIXED VULNERABILITIES OF
CWEs IN C LANGUAGE

CWE	p-value	effect-size
CWE-190	3.476e-13	0.615 (large)
CWE-20	< 2.2e-16	0.675 (large)
CWE-264	7.639e-09	0.506 (large)
CWE-399	0.003	0.301 (small)
CWE-416	0.001	0.232 (small)
CWE-476	0.018	0.204 (small)

TABLE IX
THE RESULTS OF P-VALUE TOGETHER WITH CLIFF'S $|d|$ EFFECT SIZE ON
SURVIVAL TIME FOR SELF VS NON-SELF-FIXED VULNERABILITIES OF
CWEs IN PHP LANGUAGE

CWE	p-value	effect-size
CWE-352	4.651e-13	0.661 (large)
CWE-79	9.484e-15	0.378 (medium)
CWE-264	0.002	0.206 (small)
CWE-284	0.033	0.195 (small)
CWE-200	0.046	0.136 (negligible)
CWE-89	0.032	0.077 (negligible)

TABLE X
AVERAGE SURVIVAL TIME OF SELF-FIXED (AT-SF) AND OF
NON-SELF-FIXED (AT-NSF) FOR BOTH C AND PHP LANGUAGES

Language	AT-SF (days)	AT-NSF (days)	AT-NSF/AT-SF
C	941.56	1831.47	1.94
PHP	814.86	1429.02	1.75

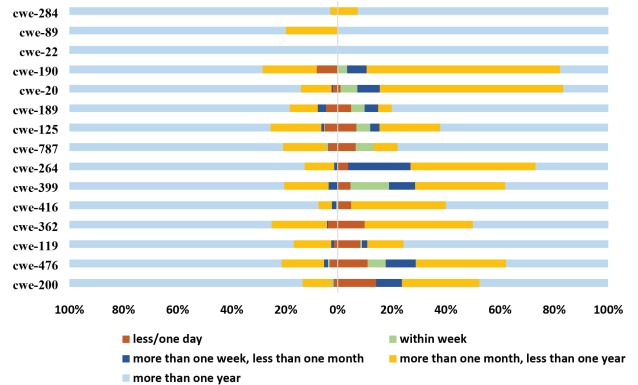


Fig. 1. Distribution of the survival time of self-fixed (right side) and of non-self-fixed (left side) CWE issues for C language

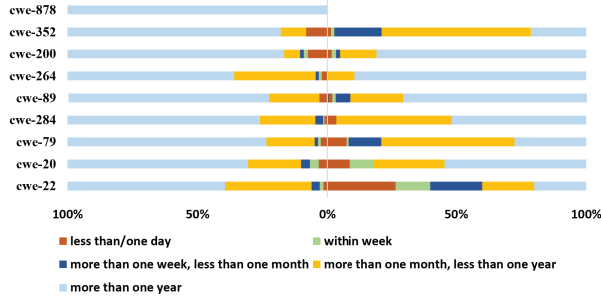


Fig. 2. Distribution of the survival time of self-fixed (right side) and of non-self-fixed (left side) CWE issues for PHP language

develop tools aimed at supporting developers in activities of vulnerability remediation. When a vulnerability is discovered, based on the vulnerability type, similar tools could be useful for identifying the best candidates for resolving it in the possible shortest time. Automated tools for bug assignment can benefit from the findings of this study, as they could integrate activity-based and location-based features [1], [36] to enhance their effectiveness with vulnerability-related bugs. Activity-based features can model information related to developers' previous experience with resolved and modified files. Similarly, location-based features can model the developers who recently and frequently modified buggy files and lines. Such tools could also leverage our approach to help project managers identify the developers responsible for the introduction of a vulnerability and more easily involve them in the definition (and the implementation) of the remediation strategy. In addition, the obtained results also complement the ones achieved in previous work [27], [5], as they allow a more in-depth understanding of which are the vulnerabilities that could be likely fixed by the developer(s) who introduced them and which vulnerability types more likely require the contribution of security experts. Interestingly, the fact that a vulnerability is widespread does not entail that developers are able to fix it. This leads to consider that the knowledge about vulnerability fixing is still not uniformly diffused among developers. This could be also due to the different availability of resources for the different vulnerabilities which makes it more difficult handling some vulnerabilities rather than others. It urges to build a more mature and complete body of knowledge for all the existing vulnerabilities and a more uniform training among developers.

Practical implications for practitioners. Practitioners can also benefit from the obtained results to better plan maintenance activities. For example, software development teams, based on their priority, can decide which vulnerabilities can be fixed in the near future by first assigning them to the developers who likely introduced them. On the one hand, this could be a quick and dirty strategy for organizations that lack a properly defined vulnerability remediation framework [2]. On the other hand, organizations that currently make use of vulnerability managing tools, can adjust their training plans

or could collect more effective resources to handle more successfully those vulnerabilities that require more time to be fixed.

V. THREATS TO VALIDITY

Threats to construct validity are related to the connection between the research questions and the objects of study. Since we identified vulnerability fixing commits by extracting them from the NVD reports, it is possible that some of the commits are not security fixes, as they may instead reference the change that introduced the vulnerability. This could represent a threat to construct validity, as we may have analyzed source code transformations that introduced the vulnerability, instead of the changes connected with its remediation. To partially mitigate this concern, two authors, experienced in vulnerability fixing, by manually checking the (i) CVE descriptions, (ii) the commit messages, and (iii) the corresponding source code modifications related to 100 randomly sampled commits, found that all the investigated commits were actually patches for the related vulnerabilities. Indeed, previous research [19] demonstrated that in almost all the cases the links referenced by NVD reports reflect fixes for the corresponding vulnerabilities. Moreover, vulnerabilities in our dataset have been grouped according to CWE categories and misclassifications could lead to erroneous results. To alleviate this issue, we mined the CVE details website in order to assign CWE categories to each considered vulnerability in our dataset. Besides, to limit the number of false positives when considering hunks, we filtered out (i) hunks that are fixed without any deletion of source code, (ii) comment lines of code (CLOC) when including deleted lines, and (iii) faulty lines that exclusively contain newline characters (e.g., "\n", "\n\n", and etc) or special marks (e.g., "(", ")", "[", "]", "{", "}", etc).

Threats to internal validity concern confounding factors that could affect our conclusions. In our study, we assume that all the changes performed in a vulnerability-fixing commit are exclusively aimed at resolving the specific vulnerability. However, developers could apply code modifications having multiple purposes into a single commit, performing a *tangled change*. Previous work [10] demonstrated that up to 15% of bug-fixing commits are tangled. However, as security fixes are very localized (i.e., affecting fewer files and having lower footprints than changes aimed at fixing non-security bugs [19]), we believe that the impact of this issue on our results would only be marginal. As a future investigation, we plan to study the impact of tangled changes on vulnerability-fixing commits. Moreover, in our study, we assume that the quality of patches is similar in self-fixed and non-self-fixed vulnerabilities. This aspect of the vulnerability fixing process needs to be more in-depth investigated in future studies.

Threats to conclusion validity concern the relation between experimentation and outcome. We used appropriate statistical procedures in order to draw our conclusions. In particular, we used non-parametric tests as the variables of interest are not well-modeled by normal distributions (as verified through the

Shapiro-Wilk test [33]). More precisely, we used the Mann-Whitney test for investigating the differences (in terms of survival time) occurring among the self and non-self-fixed vulnerabilities and used Cliff's delta effect size to investigate the magnitude of statistically significant differences.

Threats to external validity concern the generalizability of our findings. Our study is small in scale and only focuses on limited sets of projects developed in the C and PHP programming languages. For this reason, our results may be specific to the investigated subjects and further investigations are required to establish if they may apply to all the projects developed in such languages. Furthermore, we exclusively analyzed open-source projects, and it is unclear if our results may generalize to proprietary software. Besides, the set of CWEs considered in this study is not exhaustive. However, we focused our investigation on (i) the vulnerabilities belonging to the most dangerous software weaknesses and (ii) the most widespread CWEs in the analyzed projects.

VI. RELATED WORK

In the literature, there have been studies investigating the evolution of vulnerabilities and the corresponding remediations in the software systems. In this context, Shahzad *et al.* [32], presented a large scale study of various aspects associated with software vulnerabilities during their life cycle. They found that patching vulnerabilities in closed-source software is usually faster compared to open-source software. Similarly, Li and Paxson [19] conducted an analysis of various aspects of the vulnerability patching development life cycle, observing that many vulnerabilities survive in the system for a long time before their fixes. Ozment *et al.* [25] investigated the evolution of vulnerabilities in the OpenBSD operating system over time, observing that it took on average 2.6 years for a release version to remedy half of the known vulnerabilities. Researchers also explored Android-related vulnerabilities and showed that although most apps exhibit critical vulnerabilities [7], the time interval between the introduction of a vulnerability and the corresponding patch is on average 724 days [20]. Frei *et al.* [8] analyzed more than 80,000 security advisories published between 1995 and 2006, and provided insights on how vulnerabilities are handled by investigating discovery dates, disclosure dates, as well as the exploit and patch availability dates. Huang *et al.* [12] investigated the delays that could exist between the discovery of vulnerabilities and the release of the corresponding patches.

Other studies in this field more closely explored how developers manage software vulnerabilities in different ways. A recent study by Canfora *et al.* [5] found that vulnerabilities are fixed by involving a higher number of contributors and longer discussions, but they are resolved in less time compared to canonical issues. Recently, Alomar *et al.* [2] found that vulnerability remediation not only lacks proper attention, it can also mute any progress made in vulnerability discovery techniques. Their study's participants mentioned that fixing a vulnerability could be as hard as finding it. Bandra *et al.* [3] studied the commits fixing a vulnerability to understand the

success of the remediation. Piantadosi *et al.* [27], investigated vulnerability fixes on Apache HTTP and Apache Tomcat projects. Their analysis showed that vulnerability issues are (i) fixed more by experienced developers who did not author them, (ii) mostly fixed before they are publicly disclosed with the exception of partially patching in some cases, and (iii) with few modifications when patching. Researchers in [39] conducted a large-scale empirical study to characterize and understand developers' interactions during security activities, including both vulnerabilities introducing and fixing activities. They found that the majority of projects are not hero-centric when evaluating developers' contributions to security activities. Moreover, different projects include different types of interactions among developers, and these interactions correlate with the software quality of projects. Meneely *et al.* [23] studied the relationship between collaborative reviews and vulnerabilities. They showed that developers with less experience in vulnerability fixing reviews are likely to miss associated vulnerabilities, as they may suffer from a lack of security experience and collaborator familiarity.

Differently from these previous studies, we analyze the vulnerability self-fixing phenomenon for 17 different categories of vulnerabilities (CWEs) and found the specific CWE categories that could be more or less self-fixing prone. Besides, we compared the survival time of self and non-self fixed issues in C and PHP projects.

VII. CONCLUSION

In this paper, we empirically investigated the phenomenon of self-fixed vulnerabilities. Specifically, we analyzed the extent of this phenomenon over 17 different types of software weaknesses (CWEs) and 1752 vulnerability fixing commits related to C and PHP projects. Our results showed that the majority of fixed vulnerabilities were resolved by developers who did not introduce them. However, for the majority of the investigated CWEs, self-fixed vulnerabilities are likely resolved faster than non-self-fixed ones. We also found that, in C projects, while CWE-190 vulnerabilities are more likely to be self-fixed, vulnerabilities related to improper calculation or conversion of numbers are fixed more and faster by other developers. Instead, in PHP projects, developers tend to self-fix more SQL injection and CSRF vulnerabilities. In particular, while self-fixed CWE-89 vulnerabilities are not resolved much faster than non-self-fixed ones, self-fixed CWE-352 vulnerabilities are resolved about 9 times faster than non-self-fixed ones. However, vulnerabilities depending on improper access restrictions to resources are repaired more and faster by developers different from those who introduced them.

In the future, we plan to extend this study by investigating a self-fixed vulnerability in other programming languages. We also plan to (i) expand the considered software weakness in future analysis and (ii) investigate the quality of self-fixed and non-self-fixed vulnerability patches. Another future direction is the design of vulnerability assignment recommendation systems based on our results and their evaluation with developers and project managers.

REFERENCES

- [1] B. Alkhazi, A. DiStasi, W. Aljedaani, H. Alrubaye, X. Ye, and M. W. Mkaouer. Learning to rank developers for bug report assignment. *Applied Soft Computing*, 95:106667, 2020.
- [2] N. Alomar, P. Wijesekera, E. Qiu, and S. Egelman. "you've got your nice list of bugs, now what?" vulnerability discovery and management processes in the wild. In *Sixteenth Symposium on Usable Privacy and Security ({SOUPS}) 2020*, pages 319–339, 2020.
- [3] V. Bandara, T. Rathnayake, N. Weerasekara, C. Elvitigala, K. Thilakarathna, P. Wijesekera, and C. Keppitiyagama. Fix that fix commit: A real-world remediation analysis of javascript projects. In *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 198–202. IEEE, 2020.
- [4] G. Canfora, A. Di Sorbo, S. Forootani, M. Martinez, and C. A. Visaggio. Patchworking: Exploring the code changes induced by vulnerability fixing activities. *Information and Software Technology*, 142:106745, 2022.
- [5] G. Canfora, A. Di Sorbo, S. Forootani, A. Pirozzi, and C. A. Visaggio. Investigating the vulnerability fixing process in oss projects: Peculiarities and challenges. *Computers & Security*, 99:102067, 2020.
- [6] D. A. Da Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho, and A. E. Hassan. A framework for evaluating the results of the szz approach for identifying bug-introducing changes. *IEEE Transactions on Software Engineering*, 43(7):641–657, 2016.
- [7] A. Di Sorbo and S. Panichella. Exposed! A case study on the vulnerability-proneness of google play apps. *Empirical Software Engineering*, 26(4):78, 2021.
- [8] S. Frei, M. May, U. Fiedler, and B. Plattner. Large-scale vulnerability analysis. In *Proceedings of the 2006 SIGCOMM workshop on Large-scale attack defense*, pages 131–138, 2006.
- [9] R. J. Grissom and J. J. Kim. *Effect sizes for research: A broad practical approach*. Lawrence Earlbaum Associates, 2nd edition edition, 2005.
- [10] K. Herzig and A. Zeller. The impact of tangled code changes. In *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*, pages 121–130, 2013.
- [11] M. K. Hossen, H. Kagdi, and D. Poshyanyk. Amalgamating source code authors, maintainers, and change proneness to triage change requests. In *Proceedings of the 22nd International Conference on Program Comprehension*, pages 130–141, 2014.
- [12] Z. Huang, M. D'Angelo, D. Miyani, and D. Lie. Talos: Neutralizing vulnerabilities with security workarounds for rapid response. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 618–635. IEEE, 2016.
- [13] G. Jeong, S. Kim, and T. Zimmermann. Improving bug triage with bug tossing graphs. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 111–120, 2009.
- [14] T. Jiang, L. Tan, and S. Kim. Personalized defect prediction. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 279–289. Ieee, 2013.
- [15] M. Jimenez, M. Papadakis, T. F. Bisseyandé, and J. Klein. Profiling android vulnerabilities. In *2016 IEEE International conference on software quality, reliability and security (QRS)*, pages 222–229. IEEE, 2016.
- [16] M. Jimenez, M. Papadakis, and Y. L. Traon. An empirical analysis of vulnerabilities in openssl and the linux kernel. In *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*, pages 105–112, 2016.
- [17] S. Kim, E. J. Whitehead, and Y. Zhang. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering*, 34(2):181–196, 2008.
- [18] S. Kim, T. Zimmermann, K. Pan, and E. J. Jr. Whitehead. Automatic identification of bug-introducing changes. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, pages 81–90, 2006.
- [19] F. Li and V. Paxson. A large-scale empirical study of security patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS '17*, pages 2201–2215, New York, NY, USA, 2017. ACM.
- [20] M. Linares-Vásquez, G. Bavota, and C. Escobar-Velásquez. An empirical study on android-related vulnerabilities. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 2–13. IEEE, 2017.
- [21] M. Linares-Vásquez, K. Hossen, H. Dang, H. Kagdi, M. Gethers, and D. Poshyanyk. Triaging incoming change requests: Bug or commit history, or code authorship? In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 451–460. IEEE, 2012.
- [22] A. Meneely, H. Srinivasan, A. Musa, A. R. Tejeda, M. Mokary, and B. Spates. When a patch goes bad: Exploring the properties of vulnerability-contributing commits. In *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 65–74. IEEE, 2013.
- [23] A. Meneely, A. C. R. Tejeda, B. Spates, S. Trudeau, D. Neuberger, K. Whitlock, C. Ketant, and K. Davis. An empirical investigation of socio-technical code review metrics and security vulnerabilities. In *Proceedings of the 6th International Workshop on Social Software Engineering, SSE 2014*, page 37–44, New York, NY, USA, 2014. Association for Computing Machinery.
- [24] P. J. Morrison, R. Pandita, X. Xiao, R. Chillarege, and L. Williams. Are vulnerabilities discovered and resolved like other defects? *Empirical Software Engineering*, 23(3):1383–1421, 2018.
- [25] A. Ozment and S. E. Schechter. Milk or wine: does software security improve with age? In *USENIX Security Symposium*, volume 6, 2006.
- [26] H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, and Y. Acar. Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 426–437, 2015.
- [27] V. Piantadosi, S. Scalabrino, and R. Oliveto. Fixing of security vulnerabilities in open source projects: A case study of apache HTTP server and apache tomcat. In *12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019, Xi'an, China, April 22-27, 2019*, pages 68–78, 2019.
- [28] B. Ray, D. Posnett, V. Filkov, and P. Devanbu. A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 155–165, 2014.
- [29] P. Runeson, M. Host, A. Rainer, and B. Regnell. *Case study research in software engineering: Guidelines and examples*. John Wiley & Sons, 2012.
- [30] F. Schuckert, B. Katt, and H. Langweg. Source code patterns of sql injection vulnerabilities. In *Proceedings of the 12th International Conference on Availability, Reliability and Security*, pages 1–7, 2017.
- [31] H. Shahriar and H. M. Haddad. Rule-based source level patching of buffer overflow vulnerabilities. In *2013 10th International Conference on Information Technology: New Generations*, pages 627–632. IEEE, 2013.
- [32] M. Shahzad, M. Z. Shafiq, and A. X. Liu. A large scale exploratory analysis of software vulnerability life cycles. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 771–781. IEEE, 2012.
- [33] S. S. Shapiro and M. B. Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, 52(3/4):591–611, 1965.
- [34] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? *ACM sigsoft software engineering notes*, 30(4):1–5, 2005.
- [35] J. Tan, D. Feitosa, and P. Avgeriou. An empirical study on self-fixed technical debt. In *Proceedings of the 3rd International Conference on Technical Debt*, pages 11–20, 2020.
- [36] Y. Tian, D. Wijedasa, D. Lo, and C. Le Goues. Learning to rank for bug report assignee recommendation. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pages 1–10. IEEE, 2016.
- [37] R. Van Solingen, V. Basili, G. Caldiera, and H. D. Rombach. Goal question metric (gqm) approach. *Encyclopedia of software engineering*, 2002.
- [38] C. Wang, Y. Li, L. Chen, W. Huang, Y. Zhou, and B. Xu. Examining the effects of developer familiarity on bug fixing. *Journal of Systems and Software*, 169:110667, 2020.
- [39] S. Wang and N. Nagappan. Characterizing and understanding software developer networks in security development. *arXiv preprint arXiv:1907.12141*, 2019.
- [40] T. Ye, L. Zhang, L. Wang, and X. Li. An empirical study on detecting and fixing buffer overflow bugs. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 91–101. IEEE, 2016.
- [41] S. Zaman, B. Adams, and A. E. Hassan. Security versus performance bugs: a case study on firefox. In *Proceedings of the 8th working conference on mining software repositories*, pages 93–102. ACM, 2011.