Characterizing and Understanding Software Developer Networks in Security Development

Song Wang
York University, Toronto, Canada
wangsong@yorku.ca

Nachiappan Nagappan[§]
Microsoft Research, Redmond, USA
nnagappan@acm.org

***Abstract*—To build secure software, developers often work together during software development and maintenance to find, fix, and prevent security vulnerabilities. Examining the nature of developer interactions in security development can provide valuable insights for improving current practices.**

In this work, we first conduct a large-scale empirical study to mine developer interactions in security development regarding their security introducing and fixing activities on a benchmark dataset, which involves more 1.8M commits from nine large-scale open-source software projects. We then build software developer networks with the identified developer interactions and conduct network analysis to characterize and understand security development. For our analysis, we first study the interaction patterns between developers. Second, we characterize the nature of developer interaction in security development in comparison to developer interaction in non-security development. Then, we explore the relation between developer interaction and the quality of projects regarding security. Among our findings we identify that: the dominating interaction patterns among developers in the security and non-security development are different, which may suggest the needs of differing social and communication support for security and non-security development; the distribution of interaction patterns has a correlation with the quality of software projects; different from general software development, most of the projects are non hero-centric regarding security development. We believe the findings from this study can help developers understand how vulnerabilities originate and evolve under the interaction of developers and further improve software maintenance.

***Index Terms*—Security analysis, social network analysis, developer network, developer interaction**

I. Introduction

Building reliable and secure software becomes more and more challenging in modern software development. As vulnerabilities can have catastrophic and irreversible impacts, e.g., the recent Heartbleed vulnerability (CVE-2014-0160) cost more than US$500 million to the global economy [1].

Developing secure software is a team effort, developers work together to find, fix, and prevent security vulnerabilities and during which they form implicit collaborative developer networks [2]–[6]. Understanding the structure of developer interaction in security development can be helpful for accelerating security development management tasks and further building more secure software.

[§]Dr. Nachiappan Nagappan was with Microsoft Research when this work was done. He is currently at Facebook.

Along this line, many developer network-related analysis studies have been proposed to deal with problems in real-world security practice such as vulnerabilities prediction [2], [6], exploring the impact of human factors on security vulnerabilities [3], [7], and monitoring vulnerabilities [8], [9]. Most of the existing studies build developer social networks with a single type of developer interactions, e.g., developers have co-changed/co-commented files that contain security vulnerabilities [2], [3], [6], [9]. However, during the life cycle of a security vulnerability, developers interact with each other via multiple ways. For example, as shown in Figure 1, developers `d1` and `d2` introduced the security vulnerability `s1` via commits `c1` and `c2`; `s1` was later fixed by developer `d3` and `d4` via commits `f1` and `f2`. Security vulnerability `s2` was introduced via commit `c3` and fixed via commit `f3` by the same developer `d5`. Examining the nature of developer interactions in security development including both security introducing and security fixing activities can provide insights for improving current security practices.

In this paper, to characterize and understand developer interactions in security development, we first conduct a large-scale empirical study to mine developer interactions in security development regarding security introducing and fixing activities on a benchmark dataset about developers' interactions during their security, which involves 1.8M commits from nine large-scale open-source projects including operation systems, compilers, PHP interpreter, Android platform, and JavaScript engine, etc. We further build software developer networks with the identified developer interactions and conduct network analysis to characterize and understand security development. For our analysis, we first explore whether there exist dominating interaction patterns between developers across our experimental projects, after that we study how the distribution of developer interaction patterns changes in different projects over time. Second, we characterize the nature of developer interaction in security development in comparison to developer interaction in non-security development (i.e., introducing and fixing non-security bugs). We then explore the potential relation between developer interaction patterns and software quality regarding security aspect. Following existing studies [10], [11], we use "security density" (i.e., dividing the number of security vulnerability by the number of submitted commits) to measure software quality regarding security. In addition, we also examine whether the prevalent hero-centric development

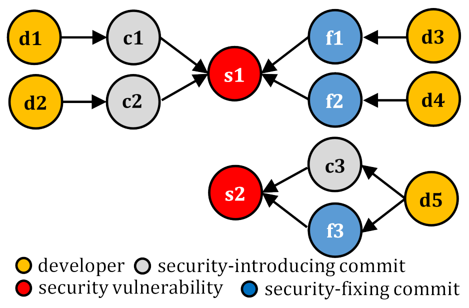


Fig. 1: An example security developer network.

phenomenon (i.e., 80% or more of the contributions are made by the 20% of the developers) in general software development also holds in security development. This paper makes the following contributions:

- We conduct the first study to analyze developer interactions in developer networks built on different types of developer interactions in security development.
- We show that developers do share common interaction patterns in the security and non-security development, while the dominating interaction patterns are different, which may suggest the needs of differing social and communication support for security and non-security development.
- We examine that developer interaction is correlated with the quality of a software project regarding security vulnerability density, which shows the potential practical value of developer interactions in monitoring software quality.
- We confirm that all experimental projects are hero-centric regarding non-security activities, while most (eight out of nine) experimental projects are non hero-centric in security development.
- We provide a benchmark dataset[1] about developer interaction during their security and non-security development, which involves 1.8M commits from nine large-scale open-source projects and could be used to facilitate future research.

The rest of this paper is organized as follows. Section II presents the background. Section III describes the methodology to collect security and non-security related commits for building developer networks. Section IV describes our approach to building developer networks. Section V discussed our research questions. Section VI presents the result of our empirical studies. Section VII discusses the threats to the validity of this work. Section VIII presents related studies. Section IX concludes this paper.

II. Background

A. Version-Control Systems

Version-control systems (VCS) are widely used in modern software development to coordinate developers' incremental contributions to a common software system. A VCS stores the entire source-code change history in the form of atomic change sets, called commits, which contain information about the changed code, the committers, and the timestamp of commits, etc. Git is one of the most popular VCSs, which has been adopted by more than 57M open-source projects and used by more than 20M developers[2] globally. Git's unique features make it especially appropriate for mining invaluable information to better understand software process [12], [13]. For example, Git can track the history of lines as they are modified. By using the `git blame` feature, we can track the modification history of each line in a commit.

In this work, we collect software security history data from nine projects that are maintained by Git to explore the developer interaction structures during their security activities (see Section III).

B. Developer Security Network

Developer interactions during developers' security development (including security fixing and introducing activities) enable us to identify collaborative relationships between developers. The developer relationships can be described by a network, in which nodes represent developers and edges represent interactions between developers.

In this study, a network can be formalized as a graph $G = (V, E)$, where V is a set of vertices and E is a set of edges, denoted by $V(G)$ and $E(G)$, respectively. An edge $e \in E$ is denoted as $e = v, u$, where v is the origin node and u is the destination node from V. Graph edges are directed with different meanings.

Different from most of existing developer social network studies [14], [15], in which $v \in V$ is a developer, and $e \in E$ represents a particular form of developer interactions, e.g., fixed bugs together [16]–[19], co-changed files [2], [6], [8], [9], [20], [21], worked on the same project [22], or have communicated via email [23], etc., we consider a $v \in V$ in a developer security network may have three different types, i.e., developer, security-fixing commit, and security-introducing commit. Consequently, a $e \in E$ has also have three different types of meanings, i.e., a developer introduces a security vulnerability via a security-introducing commit, a developer fixes a security vulnerability via a security-fixing commit, a security-fixing commit fixes the vulnerability introduced by a security-introducing commit.

III. Data Collection Methodology

A. Subject Projects

We selected nine open-source projects from existing studies [14], [24]–[27], listed in Table I, to explore developer interaction in security activities. The projects vary by the following dimensions: (a) size (lines of source code from 20K to over 17M, number of developers from 604 to 19K), (b) age (days since first commit), (c) programming language (C/C++, Java, PHP, and JavaScript), (d) application domain (operating system, compiler, PHP interpreter, Android platform, and JavaScript engine, etc.), and (e) VCS used (Git, Subversion). For each project, we extracted its code repository, and all

[1]https://zenodo.org/record/4744903

[2]https://en.wikipedia.org/wiki/GitHub

TABLE I: Experimental projects in this study. **Dev** is the number of developers. **Fix** is the number of commits that fixed security or non-security issues. **Intro** is the number of commits that introduced security or non-security issues.

Project	Language	LastCommitDate	#Commit	#Dev	#CVE	Security Vulnerability			Non-Security Bugs		
						Fix	Intro	Dev	Fix	Intro	Dev
FFmpeg	C/C++	2018/11/05	92.3K	1.7K	308	810	1.0K	199 (11.62%)	16.0K	26.5K	1.1K (66.43%)
Freebsd	C/C++	2018/11/05	255.9K	766	341	2.6K	4.1K	386 (50.39%)	35.7K	66.5K	604 (78.85%)
Gcc	C/C++	2018/11/05	165.5K	604	6	575	1.3K	200 (33.11%)	15.8K	29.9K	506 (83.77%)
Nodejs	JS	2018/11/05	24.4K	2.6K	48	252	402	105 (3.98%)	4.7K	10.5K	1.3K (49.32%)
Panda	C/C++	2018/11/05	52.6K	1.2K	24	557	1.1K	230 (18.85%)	9.1K	17.1K	838 (68.69%)
Php	C/C++	2018/11/05	109.4K	911	588	979	1.3K	165 (18.11%)	25.6K	48.2K	663 (72.78%)
Qemu	C/C++	2018/11/05	64.8K	1.5K	261	789	1.5K	263 (18.03%)	12.1K	23.9K	1.0K (70.12%)
Linux	C/C++	2018/11/05	796.0K	19.4K	2.2K	10.3K	17.1K	3.7K (19.04%)	174.6K	313.8K	14.0K (72.54%)
Android	Java	2018/11/05	377.8K	2.9K	1.7K	2.4K	2.5K	496 (16.88%)	70.1K	128.9K	2.1K (72.57%)

the historical code commits hosted in GitHub on Nov. 5th 2018. Details of our approach to collecting the commits that introduce or fix security vulnerabilities and non-security bugs are as follows.

B. Vulnerabilities Related Commits Collection

*1) **Collecting Security Vulnerability Fixing Commits**:* Our data collection of security vulnerability fixing commits starts from the National Vulnerability Database (NVD) [28], a database provided by the U.S. National Institute of Standards and Technology (NIST) with information pertaining to publicly disclosed software vulnerabilities. NVD contains entries for each publicly released vulnerability. These vulnerabilities are identified by CVE (Common Vulnerabilities and Exposures) IDs [29]. When security researchers or vendors identify a vulnerability, they can request a CVE Numbering Authority to assign a CVE ID to it. Upon public release of the vulnerability information, the summarization the vulnerability, links to relevant external references (such as security fixing commits and issue reports), list of the affected software, etc., will be added to the CVEs. We first extracted all the public CVEs of each experimental subject on Nov. 5th 2018. We then crawled the Git commit links to identify and clone the corresponding Git source code repositories and collected security fixes using the commit hashes in the links. Note that, we also find that some of the external references only contain the bug/issue report links, e.g., the external reference of security vulnerability CVE-2018-14609[3] does not contain the security fixing commits instead it shows the bug report ID[4]. For these security vulnerabilities, we used the fixing commits of these bugs as the security fixing commits. To collect the fixing commits of these bugs, we consider commits whose commit messages contain the bug report ID as the fixing commits by following existing studies [27], [30].

As reported in existing studies [31], [32], not all security vulnerability have CVE identifiers, around 53% of vulnerabilities in open source libraries are not disclosed publicly with CVEs [33], [34]. To cover all possible vulnerabilities, we used the heuristical approaches proposed by Zhou et al. [33], to identify the security fixing commits. Specifically, we used the

[3]https://nvd.nist.gov/vuln/detail/CVE-2018-14609
[4]https://bugzilla.kernel.org/show_bug.cgi?id=199833

Algorithm 1 Grouping Fixing Commits

```
Require:
    Fixing commit set $C$;
    Query fixing commit $q$;
    Commit message similarity threshold $thres_s$;
    Fixing location overlap rate threshold $modif_o$;
Ensure:
    A list of grouped fixing commit $D$;
1: for each commit $r$ in $C$ and $q$ do
2:     Extract commit messages and compute the similarity $message_s$;
3:     Extract modified files and compute the overlap rate $modif_o$;
4:     if $message_s > thres_s$ and $modif_o > 0$ then
5:         put $r$ in $D$
6:     end if
7: end for
```

regular expression rules listed in their Table 1, which included possible expressions and keywords related to security issues.

*2) **Grouping Security Fixing Commits**:* We find that some of the security fixing commits are made for fixing the same security vulnerability. For example, to fix security vulnerability CVE-2018-10883[5], developers have made two commits. Identifying fixing commits that belong to the same security vulnerability could provide us valuable information about how vulnerabilities are fixed through developer interactions. To group fixing commits, first, for fixing commits that have CVE identifiers in their commit messages, we consider fixing commits that contain the same CVE identifiers belong to the same security vulnerabilities. Second, for fixing commits that do not have CVE identifiers in their commit messages, we propose a heuristical algorithm to group them, which is described in Algorithm 1. Specifically, given two fixing commits, we group them together if the similarity of their commit messages is larger than a threshold (i.e., $message_s$) and the modification location has overlaps. Following existing study [35]–[38], we use the Cosine similarity to measure the similarity between two commit messages. We employ tf-idf [39], stop words removal (e.g., "is", "are", and "in" since these words are used in most commit messages and thus have little discriminative power) and stemming (e.g., "groups" and "grouping" are reduced to "group".) to extract string vectors

[5]https://nvd.nist.gov/vuln/detail/CVE-2018-10883

from the commit messages. For the threshold $thres_s$, we assume the ratios of collaborative fixing commits (i.e., fixing the same vulnerability) are similar between commits which have CVEs and commits that do not have CVEs. Thus for each project, we use the ratio of the collaborative fixing commits among the fixing commits that have CVEs to specify its threshold $thres_s$. We set the maximum interval between two collaborative fixing commits as six months, which is the typic length of fixing a security vulnerability [40].

3) ***Collecting Security Vulnerability Introducing Commits:*** With the above security-fixing commits, we further identify the security-introducing commits by using a blame technique provided by a Version Control System (VCS), e.g., git or SZZ algorithm [30]. Following existing studies [41]–[44], we assume the deleted lines in a security-fixing commit are related to the root cause and considered as faulty lines. The most recent commit that introduced the faulty line is considered a security-introducing commit. The details of the security-introducing commits as listed in Table I. The average number of security-introducing commits of a security-fixing commit ranges from 1.03 (Android) to 2.41 (Nodejs).

C. Non-Security Bugs Related Commits Collection

To explore the difference of developer interaction structures between developers' security activities and non-security activities, we also collect general bugs (i.e., non-security).

Typically software bugs are discovered and reported to an issue tracking system such as Bugzilla and later on fixed by the developers. A bug report usually records the description, the opening and fixing date, type (bug, enhancement, feature, etc.), etc. We consider a bug report in the Bugzilla database that is labelled as a "bug" to be a general bug. However, not all the projects have well-maintained bug tracking systems, in this work, following existing studies [42]–[44] if a project's bug tracking system is not well maintained and linked, we consider changes whose commit messages contain the word "fix" and "bug" as bug-fixing commits. If a project's bug tracking system is well maintained and linked, we consider commits whose commit messages contain a bug report ID as bug-fixing commits. For each of the bug-fixing commit, we adopt the same approach as we used to identify security-introducing commits in Section III-B3. The details of non-security fixing commits and their corresponding non-security introducing commits are showed in Table I. The average number of non-security introducing commits of a non-security fixing commit ranges from 1.66 (FFmpeg) to 2.21 (Nodejs).

In Section III-B2, we group security-fixing commits that fix the same security vulnerability. For non-security bugs, we also found the same phenomenon, i.e., some of the non-security fixing commits are made for fixing the same non-security bugs. For grouping these non-security fixing commits, we reuse Algorithm 1. As described in Section III-B2, for grouping security fixing commits, we use the ratio of collaborative fixing commits (i.e., fix the same security vulnerability) that have CVE identifiers to set the threshold $thres_s$ of a specific project. However, for non-security fixing commits, not all projects have well-maintained bug tracking systems, for some projects (e.g., Linux), we cannot use bug report ID to specify $thres_s$. Thus, we randomly pick and manually check 100 pairs of collaborative fixing commits on each the subject project, we use the average Cosine similarity value to set $thres_s$ in Algorithm 1 to group non-security fixing commits.

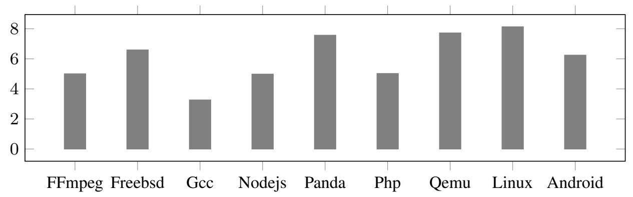


Fig. 2: The overlap rate (in percentage) of non-security introducing commits and security introducing commits.

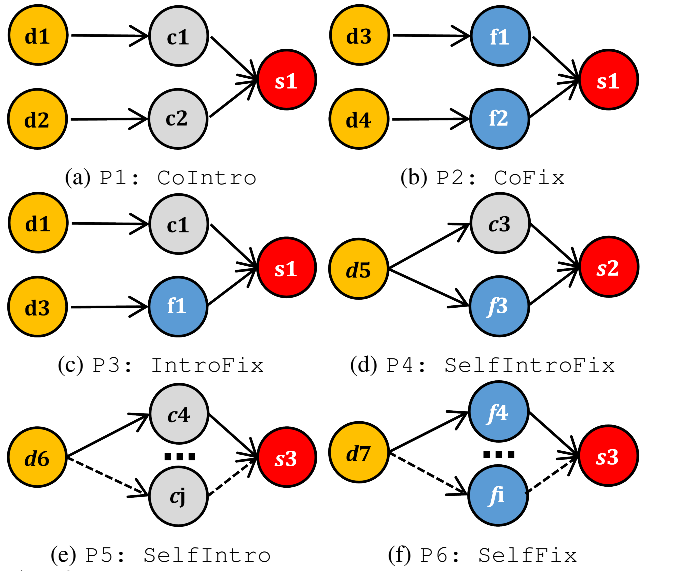
With the above non-security fixing commits, we further identify the non-security introducing commits by using a blame technique as we described in Section III-B3. For non-security introducing commits and security introducing commits, we do not handle the overlaps, since it's possible that a security vulnerability and non-security bug can be introduced by the same introducing commit. In this work, we use overlap rate to measure the overlap level between two datasets. We define the **overlap rate** between datasets A and B as $\frac{A \cap B}{A \cup B}$. Figure 2 shows the overlap rates of non-security introducing commits and security introducing commits in the experimental projects. As we can see from the figure, the overlap rates of all experimental projects are lower than 10%, which suggests that security vulnerability and non-security bugs usually have different introducing commits.

On average, the ratio for security fixing commits is 7.2% and the ratio for non-security fixing commits is 8.3%, which is consistent with the finding from an existing study [45], that 9% of bug fixes were bad across three Java projects.

D. Identifying Distinct Developers

To build the developer security network, we need to obtain the developer information of security-fixing and security-introducing commits. In Git, for every pushed commit, Git maintains the user who did the commit, i.e., committer. Git computes the committer out of the Git configuration parameters 'user.name' and 'user.email'. Thus, by retrieving a commit, we can easily obtain its committer information. However, Git also allows users to change their profiles, which introduces the alias issue of developers in mining open-source [23], [46], i.e., a developer may have different emails/names. To solve this challenge, we use the aliases unmasking algorithms proposed in [23] to identify distinct developers.

In total, we have around 45K distinct developers from the nine experimental projects, details are listed in Table I. Overall, the percentage of developer that involved in security activities ranges from 3.98% to 50.39%, while the percentage of developer that involved in non-security activities ranges from 49.32% to 83.77%.



(a) `P1: CoIntro` (b) `P2: CoFix` (c) `P3: IntroFix` (d) `P4: SelfIntroFix` (e) `P5: SelfIntro` (f) `P6: SelfFix`

Fig. 3: The meta interactions between developers during their security activities.

IV. Building Sofware Developer Network

With the collected security-fixing, security-introducing commits, and corresponding developers, in this section we present our approach to building developers networks.

In this study, we only focus on meta patterns between developers during their security activities, i.e., at most one node exists between a developer and a security vulnerability. We first identify developer's meta interactions during the security activities including both introducing and fixing security vulnerabilities. Specifically, in order to explore developer interactions, we capture three possible meta interactions between two developers, i.e., two developers introduce the same security vulnerability (`CoIntro`), two developers fix the same security vulnerability (`CoFix`), a security vulnerability is introduced by a developer and fixed by another developer (`IntroFix`), which are showed in Figure 3 from 3a to 3c. In addition, we also collect the meta interactions of a single developer, i.e., a security vulnerability is introduced and fixed by a single developer (`SelfIntroFix`), a security vulnerability is introduced by multiple commits of a single developer and fixed by other developers (`SelfIntro`), and a security vulnerability is fixed by multiple commits of a single developer and is introduced by other developers (`SelfFix`), which are showed in Figure 3 from 3d to 3f.

For each subject project listed in Table I, we collect all these meta paths and then build a security network by connecting meta paths together.

V. Research Questions

Our experimental study is designed to answer the following research questions.

RQ1. What are the distributions of developers in security and non-security activities?

Software security vulnerability and bugs are introduced and fixed by developers, in this RQ, we aim to explore the basic distribution of developers in security and non-security activities regarding fixing and introducing. For example, what is the overlap rate between developers that have ever involved in security activities and developers that have ever involved in non-security activities? What is the overlap rate between developers that have fixed security vulnerabilities and developers that have introduced security vulnerabilities?

RQ2. What are the common meta interaction patterns between two developers in security activities?

Developers interact with each other during the development of a software project. In software development, the social and organizational aspects have an impact on the individual and collective performance of the developers [47]. Along this line, in this RQ, we aim to explore the common interaction structures among developers during their security and activities regarding security fixing and security introducing across different projects, which we believe can help us gain insight into distinct characteristics of developers' security activities.

RQ3. Are the distributions of developer interaction patterns in security and non-security activities different?

To understand the difference of developers' interaction patterns during their security and non-security activities. In this RQ, we explore the nature of interaction between developers in their security activities regarding introducing and fixing security vulnerabilities in comparison to non-security activities.

RQ4. How do interaction structures among developers evolve over time?

Software team organization evolves over time [17], [48], i.e., developers may leave a project and new developers may join during the life cycle of a project, which causes the evolution of developer community. Along this line, in this RQ, we aim to explore whether the interaction structure among developers changes over time and how it evolves.

RQ5. Does the change of interaction structures have a correlation with the quality of software?

Developer social network and its evolution information have been examined could be used to predict new vulnerabilities and bugs [2], [6]. Along this line, in this RQ, we investigate whether the change of interaction structure has a correlation with the quality of software regarding the density of security vulnerabilities.

VI. Analysis Approach and Results

A. ***RQ1:*** *Distributions of Developers in Security and Non-Security Activities*

To answer this RQ, we obtain unique developers from different activities, i.e., fixing security vulnerabilities, introducing security vulnerabilities, fixing non-security bugs, and introducing non-security bugs. Given the developer sets of two activities, we calculate their overlap rates via dividing the overlapping data points by all the unique data points. Table II shows the basic overlaps between developers that have been involved in different activities. As we can see from the table, in all the projects, developers from **secFix** and **secIntro**

TABLE II: The overlap rates between developers that have been involved in different activities. **secFix** denotes developers that have made security fixing commits, **secIntro** denotes developers that have made security introducing commits, **nonSecFix** denotes developers that have made non-security fixing commits, **nonSecIntro** denotes developers that have made non-security introducing commits, and **secFix-secIntro** means the overlap rate between **secFix** and **secIntro**. The higher values with statistical significance (p-value < 0.05) are shown with an asterisk (*).

Project	secFix-secIntro (*)	secFix-nonSecFix	secFix-nonSecIntro	secIntro-nonSecFix	secIntro-nonSecIntro	sec-nonSec
FFmpeg	60.0	10.7	10.4	14.9	19.6	10.7
Freebsd	89.0	30.2	32.2	49.1	43.2	40.2
Gcc	88.1	25.6	24.5	38.8	38.4	25.6
Nodejs	63.0	5.0	5.8	7.6	10.5	5.0
Panda	65.9	17.0	18.9	21.5	32.1	17.0
Php	70.5	15.5	16.9	21.7	29.3	15.5
Qemu	67.1	16.6	18.1	20.6	28.9	16.6
Linux	66.6	16.1	18.0	21.6	29.6	16.1
Android	69.6	15.6	18.1	19.9	27.1	15.6
Average	71.1	16.9	18.1	24.0	28.8	18.0

have higher overlap rates, i.e., range from 60.0% to 89.0% and on average is 71.1%, which indicates that most of the security vulnerabilities are introduced and fixed by a core group of developers. We can also see that the overlap rates of developers from security activities and non-security activities are lower, e.g., the overlap rate of developers from **secFix** and **nonSecFix** ranges from 5.0% to 30.2% and is 16.9% on average, the overlap rate of developers from **secIntro** and **nonSecIntro** ranges from 19.6% to 38.4% and on average is 28.8%. Overall, the overlap rate from **sec** and **nonSec** is 18.6% on average, which indicates that most of the developers that are involved in security activities are different from developers that are involved in non-security activities. This may be because security issues are critical to software that require non-trivial domain expertise. Thus only a small group of developers is capable of handling security vulnerabilities, which makes the overlap rates of developers from security activities and non-security activities lower. We further conduct the Wilcoxon signed-rank test ($p < 0.05$) to compare the overlap rates among different pairs. The results suggest that the overlap rates of **secFix** and **secIntro** are significantly higher than those of other pairs.

Developers that are involved in security and non-security activities are different. Non-security bugs usually were introduced and fixed by different developers. However, security vulnerabilities were likely introduced and fixed by the same set of developers.

B. *RQ2: Common Developer Meta Interaction Patterns in Developer Security Activities*

For each subject project, with the security activity network built in Section IV, we collect the numbers and calculate the percentages of the six meta patterns, which are showed in Table III. As we can see from the figure, the six meta interaction patterns among developers exist in each of the experimental projects. The `CoIntro` and `IntroFix` patterns are dominating (i.e., the accumulated percentage is larger than 80%) across all the experimental projects. Other patterns take up around 20% of developer interactions, for example, the percentages of`SelfFix` are lower than 1% in all experimental projects. Although `CoIntro` and `IntroFix` are dominating, the percentages of them in different projects are different, i.e., range from 74.3% (Nodejs) to 94.9% (Freebsd). In addition, the percentage of interactions between developers (i.e., `CoIntro`, `CoFix`, and `IntroFix`) is much larger than that of interactions of the same developers (i.e., `SelfIntro`, `SelfFix`, and `SelfIntroFix`), which indicates the nature of software security development is teamwork.

TABLE III: The distribution of developer interaction patterns during security activities (in percentage).

Project	CoIntro	CoFix	IntroFix	SelfIntroFix	SelfIntro	SelfFix
FFmpeg	52.3	1.9	36.6	5.2	3.0	1.0
Freebsd	66.1	0.2	28.8	3.1	1.6	0.1
Gcc	58.0	10.4	18.7	9.9	2.8	0.1
Nodejs	50.9	7.4	23.4	10.7	7.3	0.3
Panda	70.3	0.7	19.6	5.1	4.0	0.3
Php	73.1	1.4	18.0	3.8	3.3	0.4
Qemu	68.0	1.8	19.3	6.6	3.9	0.5
Linux	67.1	0.5	21.9	5.8	4.5	0.2
Android	55.4	8.1	25.3	3.2	7.4	0.6
Average	62.4	3.6	23.5	5.9	4.2	0.4

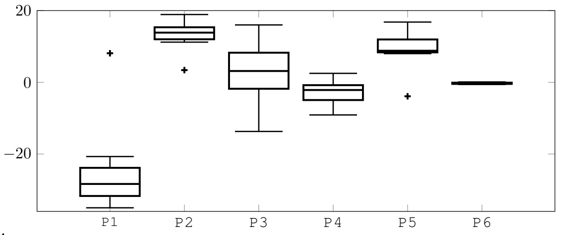


Fig. 4: The difference of the percentages of interaction patterns between security activities and non-security activities.

The percentages of meta patterns among developers vary dramatically in different projects. However, `CoIntro` and `IntroFix` patterns are dominating across all the experimental projects in developers' security activities.

C. *RQ3: Comparison of Developer Interaction Patterns between Security and Non-Security Activities*

In this RQ, we try to explore the difference of developer interactions between developers' security activities and non-security activities, which we believe can help us gain insight

TABLE IV: The distribution of developer interaction during non-security activities (in percentage).

Project	CoIntro	CoFix	IntroFix	SelfIntroFix	SelfIntro	SelfFix
FFmpeg	30.4	14.8	31.5	3.2	19.7	0.4
Freebsd	40.4	11.4	30.3	2.6	15.1	0.2
Gcc	37.3	26.1	22.4	2.4	11.7	0.2
Nodejs	59.0	26.3	9.7	1.5	3.4	0.1
Panda	39.4	4.2	35.6	3.9	16.6	0.3
Php	42.7	16.2	25.3	3.7	12.0	0.1
Qemu	35.4	19.7	28.4	4.5	11.9	0.1
Linux	32.1	15.5	33.0	3.5	15.8	0.1
Android	29.1	20.8	27.9	5.7	16.4	0.1
Average	38.4	17.2	27.1	3.4	13.6	0.2

into distinct characteristics of developers' security activities. For each subject project, we first build a non-security developer network following our approach in Section IV, then we further collect the ratios of the six meta patterns (as shown in Figure 3) in the non-security developer networks. For each meta pattern, we calculate the difference between its ratio from non-security developer network and developer network of a project. We show the detailed difference of interaction patterns between security activities and non-security activities in Figure 4. Specifically, the percentages of patterns `CoIntro` and `CoFix`, and `SelfIntro` vary dramatically across the projects in this work.

Table IV shows the distribution of the six developer interaction patterns in developers' non-security activities. Different from security activities, the dominating patterns (i.e., the accumulated percentage is larger than 80%) in non-security activities include three patterns, i.e., `CoIntro`, `IntroFix`, and `CoFix`. Note that in security activities, the percentage of `CoFix` pattern ranges from 0.2% to 10.4% and on average is 3.5%, while in non-security activities it ranges from 4.2% to 26.1% on average is 17.2%. This may indicate that fixing security vulnerability requires more domain expertise and only a small number of developers are capable to fix security vulnerabilities, thus results in less teamwork. In addition, we also find that the dominating patterns are more balanced in developers' non-security activities compared to security activities. For example, the difference of the percentages of dominating patterns in security activities ranges from 15.7% to 55.1% and on average is 38.8%, while in non-security activities, the difference ranges from 8.3% to 35.2% and on average is 21.2%.

The different dominating patterns may indicate the difference of developers' communication and social activities in security and non-security development, however neither current security or non-security development methodologies [49] could reflect such difference.

Developers have different dominating patterns in security and non-security activities, which might suggest the needs of differing social and communication support for security and non-security development.

TABLE V: The correlated patterns in each project.

Project	Correlated Patterns
FFmpeg	P1, P3, P5
Freebsd	P1, P3, P4
Gcc	P1, P2, P3, P4
Nodejs	P1, P2, P3, P4
Panda	P1, P3
Php	P1, P3, P4, P5
Qemu	P1, P3, P5
Linux	P1, P3
Android	P1, P2, P3, P4, P5

*D. **RQ4:** Evolution of Developer Interaction in Developer Security Activities*

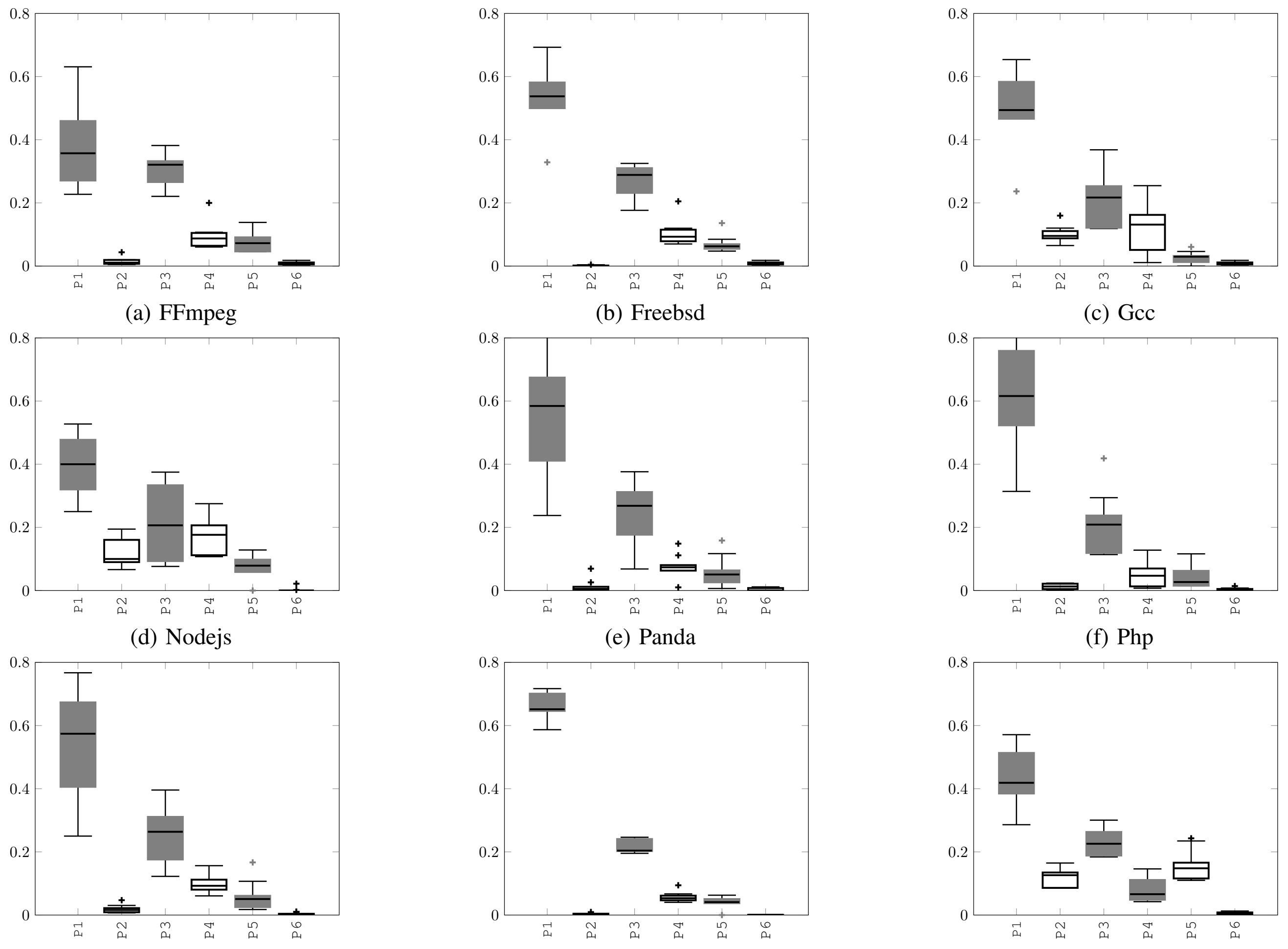
To explore the evolution of developer interactions, for each project, we collect the numbers and calculate the percentages of the six patterns that only appear in a specific year from 2007 to 2017. Specifically, given a year n ($2006 < n < 2018$), we first use the proposed approach in Section III-B and Section III-C to collect security fixing commits that happened in the year. Since the security introducing commits are derived from fixing commits, we then use the available security information in the year $n + 1$ to find the fixing commits between the years n and $n + 1$. After that, we further obtain the corresponding security introducing commits in the year n. When security fixing and introducing commits are ready, we build the developer network as described in Section II-B and further obtain the number of each interaction pattern between developers listed in Figure 3.

In total, for each pattern, we have 10 different percentage values in each project. Figure 5 shows the boxplots of the percentages of each interaction pattern in each project. Overall, the percentage of a specific pattern varies dramatically in a project over time, for example, in FFmpeg, the percentage of pattern `CoIntro` ranges from 22.7% to 63.1% in 10 years, which may be because there exist significant changes on the adopted software development processes or the project's developer team that yield dramatically different software quality. Despite the evolution of the percentages of patterns, we can observe that there exist dominating patterns in each of the projects over the 10 years. Specifically, we find that patterns `CoIntro` and `IntroFix` are dominating on each project over time.

The percentages of developer interaction patterns vary over time. While all the projects do not witness a change in terms of the dominating patterns.

*E. **RQ5:** Relation between Developer Interaction and Software Quality*

To explore the relation between developers' interactions in security development and software quality regarding security (i.e., security density), we use the data of the years from 2007 to 2017 collected in RQ4 (see Section VI-D). Following existing studies [6], [50], [51], we use the Spearman rank correlation [52] to compute the correlations between the percentages of patterns and the density of security vulnerability appeared in each year from 2007 to 2017. The closer the value



(a) FFmpeg (b) Freebsd (c) Gcc (d) Nodejs (e) Panda (f) Php (g) Qemu (h) Linux (i) Android

Fig. 5: The distribution of the number of different patterns in each project from 2009 to 2018. P1 denotes `CoIntro`, P2 denotes `CoFix`, P3 denotes `IntroFix`, P4 denotes `SelfIntroFix`, P5 denotes `SelfIntro`, and P6 denotes `SelfFix`.

of a correlation is to +1 (or -1), the higher two measures are positively (or negatively). A value of 0 indicates that two measures are independent. Values greater than 0.10 can be considered a small effect size; values greater than 0.30 can be considered a medium effect size [6]. In this work, we consider the values larger than 0.10 or smaller than -0.10 as correlated, others are uncorrelated.

Table V shows the correlated patterns in each project. As we can see, five of the six patterns are selected as correlated in at least one project. In addition, the dominating patterns `CoIntro` and `IntroFix` are selected across all projects, which suggests the practice values of using these patterns to predict the quality of projects regarding security.

Developers' interaction in security activities is correlated with the density of security vulnerabilities.

VII. Discussion

A. Heroism in Security Development

Recent studies [26], [53]–[57] show that most software projects are hero-centric projects, where 80% or more of the contributions (e.g., the number of commits) are made by the 20% of the developers. Most of existing studies explore the heroism of projects from developers' code contribution and social communication perspectives, e.g., Agrawal et al. [53] used the number of commits made by each developer to represent its contribution to a project. In this work, we try to explore the heroism in security development of a project. Specifically, we assess developers' contribution by using a specific type of commits, e.g., security fixing and security introducing, non-security fixing, and non-security introducing commits. Following existing studies [53], [54], we define a project to be hero-centric when 80% of the contributions are done by about 20% of the developers in this study.

We first examine whether a project is hero-centric when only considering a specific type of commits e.g., security fixing commits. To assess the contribution of a developer, following Agrawal et al. [53], we count the number of a specific type of commits made by each developer to represent his/her contribution to a project. We then rank developers ascendingly based on their contributions. Finally, we accumulate developers' contributions and record developers involved until 80% of

TABLE VI: The percentages of developers involved when contributing 80% of a specific type of commits. Values with a red diamond (◇) indicate that a project is non hero-centric project. **All** denotes the combination of the four types of commits.

Project	secFix	secIntro	nonSecFix	nonSecIntro	All
FFmpeg	23.1 (◇)	20.2 (◇)	3.6	5.5	3.5
Freebsd	32.1 (◇)	26.0 (◇)	13.4	11.3	11.1
Gcc	33.1 (◇)	21.1 (◇)	17.5	16.3	15.6
Nodejs	34.0 (◇)	24.7 (◇)	13.5	6.6	5.1
Panda	36.5 (◇)	25.8 (◇)	10.6	10.1	7.5
Php	21.6 (◇)	23.7 (◇)	6.3	8.2	5.7
Qemu	30.1 (◇)	22.3 (◇)	8.6	9.8	6.8
Linux	30.9 (◇)	21.6 (◇)	11.0	11.4	8.5
Android	32.7 (◇)	21.3 (◇)	11.5	11.0	8.3

TABLE VII: The overlap rates of "core developers" between different types of commits. The higher values with statistical significance (p-value < 0.05) are shown with an asterisk (*).

Project	secFix-secIntro (*)	secFix-nonSecFix	secIntro-nonSecIntro
FFmpeg	71.1	50.0	68.9
Freebsd	69.9	55.6	67.8
Gcc	69.2	32.9	43.5
Nodejs	66.7	24.4	29.2
Panda	67.0	48.0	62.5
Php	48.5	54.3	66.7
Qemu	64.6	50.0	63.8
Linux	64.8	38.0	45.3
Android	47.0	34.3	45.8
Average	63.2	43.0	55.0

the contributions are done. In addition, we also evaluate a developer's contribution via the combination of the four types of commits, i.e., security fixing, security introducing, non-security fixing, and non-security introducing.

Table VI shows the percentages of developers involved when contributing 80% of a particular type of commits. As we can see from the table, when assessing developers' contribution by using non-security fixing or non-security introducing commits or all commits together, all the projects are hero-centric, i.e., the percentages of developers involved are smaller than 20%. However, all the experimental projects are non hero-centric projects when assessing developers' contribution by using security fixing or security introducing commits, e.g., the percentage of developers involved are 36.5%, when evaluating developers' contribution by using security fixing commits in project Panda. Our finding indicates that although general software development has "heroes", i.e., a small percentage of the staff who are responsible for most of the progress on a project, software security does not have typical "heroes".

We further calculate the overlap rates of "core developers" (i.e., the set of developers that contribute 80% of a specific type of commits) between different types of commits, which are shown in Table VII. As we can see, the "core developers" from security fixing and security introducing have high overlap rates that range from 47.7% to 71.1% and on average is 63%, which is consistent with our findings in Sec VI-A. The overlap rates of the "core developers" from security commits and non-security commits, i.e., "core developers" from **secFix** and **nonSecFix**, "core developers" from **secIntro** and **nonSecFix** are lower than the rates of "core developers" only from security activities, i.e., **secFix** and **SecIntro**. This indicates that the "core developers" of security and non-security activities are different in most of the experimental projects.

All experimental projects are hero-centric regarding non-security activities, while most (eight out of nine) experimental projects are non hero-centric in security development.

B. Threats to Validity

a) Internal Validity: Threats to internal validity are related to experimental errors. Following previous work [30], [42]–[44], the process of collection security introducing or non-security introducing commits is automatically completed with the annotating or blaming function in VCS with keyword searching (e.g., "fix" and "bug" for non-security changes). It is known that this process can introduce noise [30]. The noise in the data can potentially affect the result of our study. Manual inspection of the process shows reasonable precision and recall on open source projects [44], [58]. To mitigate this threat, we use the noise data filtering algorithm introduced in [58] to remove potential noisy data.

b) External Validity: Threats to external validity are related to the generalization of our study. The examined projects in this work have a large variance regarding project types. We have tried our best to make our dataset general and representative. However, it is still possible that the nine projects used in our experiments are not generalizable enough to represent all software projects. Our approach might generate similar or different results for other projects that are not used in the experiments. We mitigate this threat by selecting projects of different functionalities (operating systems, servers, and desktop applications) that are developed in different programming languages (C, Java, and JavaScript).

In this work, all the experimental subjects are open source projects. Although they are popular projects and widely used in security research, our findings may not be generalizable to commercial projects or projects in other languages.

VIII. Related Work

A. Developer Social Network

There has been a body of work that investigated aspects of developer social networks built on developers' activities during software development [14], [59]–[66].

Lopez-Fernandez et al. [64], [65] first examined the social aspects of developer interaction during development, where developers were linked based on contributions to a common module. Bird et al. [23] investigated developer organization and community structure in the mailing list of four open-source projects and used modularity as the community-significance measure to confirm the existence of statistically significant communities. Wolf et al. [21], [67] introduced an approach to mining developer collaboration from communication repositories and they further use developer collaboration to predict software build failures. Toral et al. [66] applied social-network analysis to investigate participation inequality in the Linux mailing list that contributes to role separation between core and peripheral contributors. Hong et al. [17]

and Zhang et al. [68] explored the characteristics of developer social networks built on developer interactions in bug tracking systems and how these networks evolve over time. Surian et al. [69] extracted developer collaboration patterns from a large developer collaborations network extracted from SourceForge.Net, where developers are considered connected if both of them are listed as contributors to a project. Jeong et al. [16] and Xuan et al. [18] leveraged network metrics mined from social networks built in bug tracking systems to recommend developers for fixing new bugs. Surian et al. [22] used developer collaboration network extracted from Sourceforge.Net to recommend a list of top developers that are most compatible based on their programming language skills, past projects and project categories they have worked on before for a developer to work with. Researchers have also built social networks based on developers' security activities, i.e., have co-changed files that contain security vulnerabilities to predict new vulnerabilities [2], [6], exploring the impact of human factors on security vulnerabilities [3], [7], and monitoring vulnerabilities [8], [9].

Most of the above studies construct developer networks based on a particular form of developer collaboration e.g., co-changed files, co-commented bugs, and co-contributed projects, etc., from bug tracking systems, mailing lists, or project contribution lists. These developer networks are homogeneous, which have merely one type of node (developers) and one type of link (a particular form of developer collaboration). Wang et al. [19] and Zhang et al. [70] leveraged heterogeneous network analysis to mined different types of developer collaboration patterns in bug tracking system and further used these different collaborations to assist bug triage.

Our work differs in two ways from most of these prior studies: (1) We study developers' social interactions in security activities. (2) We explore different types of developer interactions during their security activities, which is more complex and with richer information.

B. Security Vulnerability Analysis

There are many studies to explore, analyze, and understand software security vulnerabilities [40], [41], [71]–[80].

Frei et al. [81] examined how vulnerabilities are handled with regard to information about discovery date, disclosure date, as well as the exploit and patch availability date in large-scale by analyzing more than 80,000 security advisories published between 1995 and 2006. Walden et al. [82] provided a vulnerability dataset for evaluating the vulnerability prediction effectiveness of two modelling techniques, i.e., software metrics based and text mining based approaches. Medeiros et al. [78] examined the performance of software metrics on classifying vulnerable and non-vulnerable units of code. Yang et al. [79] leveraged software network to evaluate structural characteristics of software systems during their evolution. Decan et al. [74] and Shahzad [75] presented a large scale study of various aspects associated with software vulnerabilities during their life cycle. Ozment et al. [77] 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. Perl et. al. [41] analyzed Git commits that fixed vulnerabilities to produce a code analysis tool that assists in finding dangerous code commits. Xu et. al. [80] developed a method for identifying security patches at the binary level based on execution traces, providing a method for obtaining and studying security patches on binaries and closed-source software. Li et al. [40] conducted an analysis of various aspects of the patch development life cycle. There also existed some other studies that explored the characteristics of software general bugs [83]–[85].

In this work, we propose the first study to characterize and understand developers' interaction by considering their activities in introducing and fixing security vulnerabilities by analyzing developer networks built on their security activities.

IX. Conclusion

This work conducts a large-scale empirical study to mine and build developer networks for characterizing and understanding developer interaction in security development, which involves 1.8M commits from nine large-scale open-source software projects. For our analysis, we first study the interaction patterns between developers. Second, we characterize the nature of developer interaction in security activities in comparison to developer interaction in non-security activities (i.e., introducing and fixing non-security bugs). Then, we explore the relation between developer interaction and the quality of projects regarding security. In addition, we also examine whether the prevalent hero-centric development phenomenon (i.e., 80% or more of the contributions are made by the 20% of the developers) in general software development also holds in security development. Our empirical studies show that: the dominating interaction patterns among developers in the security and non-security developments are different, which might suggest the needs of differing social and communication support for security and non-security development; the distribution of interaction patterns has a correlation with the quality of software projects; different from general software development, most of the projects are non hero-centric regarding security development. We believe the findings from this study can help developers understand how vulnerabilities originate and evolve under the interaction of developers and further improve software maintenance.

As future work, we plan to leverage developer interaction information to further improve software security practice by predicting future potential vulnerabilities, estimating the effort (e.g., number of developers) required to fix new security vulnerabilities, and recommending appropriate developers to fix new security vulnerabilities.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for their feedback which helped improve this paper. This work is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

[1] "Heartbleed," http://heartbleed.com/, 2018.

[2] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *TSE'11*, vol. 37, no. 6, pp. 772–787, 2011.

[3] A. Meneely and L. Williams, "Secure open source collaboration: an empirical study of linus' law," in *CCS'09*, 2009, pp. 453–462.

[4] ——, "Socio-technical developer networks: Should we trust our measurements?" in *ICSE'11*, 2011, pp. 281–290.

[5] 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 *ESEM'13*, 2013, pp. 65–74.

[6] T. Zimmermann, N. Nagappan, and L. Williams, "Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista," in *ICST'10*, 2010, pp. 421–428.

[7] A. Meneely and L. Williams, "Strengthening the empirical analysis of the relationship between linus' law and software security," in *ESEM'10*, 2010, p. 9.

[8] S. Trabelsi, H. Plate, A. Abida, M. M. B. Aoun, A. Zouaoui, C. Missaoui, S. Gharbi, and A. Ayari, "Mining social networks for software vulnerabilities monitoring," in *NTMS'15*, 2015, pp. 1–7.

[9] A. Sureka, A. Goyal, and A. Rastogi, "Using social network analysis for mining collaboration data in a defect tracking system for risk and vulnerability analysis," in *ISEC'11*, 2011, pp. 195–204.

[10] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *ICSE'05*, 2005, pp. 284–292.

[11] P. Knab, M. Pinzger, and A. Bernstein, "Predicting defect densities in source code files with decision tree learners," in *MSR'06*, 2006, pp. 119–125.

[12] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu, "The promises and perils of mining git," in *MSR'09*, 2009, pp. 1–10.

[13] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The promises and perils of mining github," in *MSR'14*, 2014, pp. 92–101.

[14] M. Joblin, W. Mauerer, S. Apel, J. Siegmund, and D. Riehle, "From developer networks to verified communities: a fine-grained approach," in *ICSE'15*, 2015, pp. 563–573.

[15] A. Jermakovics, A. Sillitti, and G. Succi, "Mining and visualizing developer networks from version control systems," in *CHASE'11*, 2011, pp. 24–31.

[16] G. Jeong, S. Kim, and T. Zimmermann, "Improving bug triage with bug tossing graphs," in *FSE'09*, 2009, pp. 111–120.

[17] Q. Hong, S. Kim, S. Cheung, and C. Bird, "Understanding a developer social network and its evolution," in *ICSM'11*, 2011, pp. 323–332.

[18] J. Xuan, H. Jiang, Z. Ren, and W. Zou, "Developer prioritization in bug repositories," in *ICSE'12*, 2012, pp. 25–35.

[19] S. Wang, W. Zhang, Y. Yang, and Q. Wang, "Devnet: exploring developer collaboration in heterogeneous networks of bug repositories," in *ESEM'13*, 2013, pp. 193–202.

[20] M. Pinzger, N. Nagappan, and B. Murphy, "Can developer-module networks predict failures?" in *FSE'08*, 2008, pp. 2–12.

[21] T. Wolf, A. Schröter, D. Damian, L. D. Panjer, and T. H. Nguyen, "Mining task-based social networks to explore collaboration in software teams," *IEEE Software'09*, vol. 26, no. 1, pp. 58–66, 2009.

[22] D. Surian, N. Liu, D. Lo, H. Tong, E.-P. Lim, and C. Faloutsos, "Recommending people in developers' collaboration network," in *WCRE'11*, 2011, pp. 379–388.

[23] C. Bird, A. Gourley, P. Devanbu, M. Gertz, and A. Swaminathan, "Mining email social networks," in *MSR'06*, 2006, pp. 137–143.

[24] T. T. Dinh-Trong and J. M. Bieman, "The freebsd project: A replication case study of open source development," *TSE'05*, vol. 31, no. 6, pp. 481–494, 2005.

[25] C. Izurieta and J. Bieman, "The evolution of freebsd and linux," in *ISESE'06*, 2006, pp. 204–211.

[26] A. Mockus, R. T. Fielding, and J. D. Herbsleb, "Two case studies of open source software development: Apache and mozilla," *TOSEM'02*, vol. 11, no. 3, pp. 309–346, 2002.

[27] Y. Tian, J. Lawall, and D. Lo, "Identifying linux bug fixing patches," in *ICSE'12*, 2012, pp. 386–396.

[28] U. N. I. of Standards and Technology, "National vulnerability database," https://nvd.nist.gov/home.cfm, 2018.

[29] M. Corporation, "Common vulnerabilities and exposures," https://cve.mitre.org/, 2018.

[30] S. Kim, T. Zimmermann, K. Pan, E. James Jr *et al.*, "Automatic identification of bug-introducing changes," in *ASE'06*, 2006, pp. 81–90.

[31] D. Wijayasekara, M. Manic, J. L. Wright, and M. McQueen, "Mining bug databases for unidentified software vulnerabilities," in *ICHSI'12*, 2012, pp. 89–96.

[32] S. E. Ponta, H. Plate, A. Sabetta, M. Bezzi, and C. Dangremont, "A manually-curated dataset of fixes to vulnerabilities of open-source software," in *MSR'19*, 2019.

[33] Y. Zhou and A. Sharma, "Automated identification of security issues from commit messages and bug reports," in *FSE'17*, 2017, pp. 914–919.

[34] "Sourceclear," https://www.sourceclear.com/, 2018.

[35] J. Wang, M. Li, S. Wang, T. Menzies, and Q. Wang, "Images don't lie: Duplicate crowdtesting reports detection with screenshot information," *IST'19*, 2019.

[36] J. Wang, Q. Cui, Q. Wang, and S. Wang, "Towards effectively test report classification to assist crowdsourced testing," in *ESEM'16*, 2016, p. 6.

[37] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of duplicate defect reports using natural language processing," in *ICSE'07*, 2007, pp. 499–510.

[38] H. Rocha, M. T. Valente, H. Marques-Neto, and G. C. Murphy, "An empirical study on recommendations of similar bugs," in *SANER'16*, vol. 1, 2016, pp. 46–56.

[39] I. H. Witten, E. Frank, M. A. Hall, and C. J. Pal, *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2016.

[40] F. Li and V. Paxson, "A large-scale empirical study of security patches," in *CCS'17*, 2017, pp. 2201–2215.

[41] 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 *CCS'15*, 2015, pp. 426–437.

[42] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *MSR'05*, vol. 30, no. 4, 2005, pp. 1–5.

[43] 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," *TSE'17*, vol. 43, no. 7, pp. 641–657, 2017.

[44] T. Jiang, L. Tan, and S. Kim, "Personalized defect prediction," in *ASE'13*, 2013, pp. 279–289.

[45] Z. Gu, E. T. Barr, D. J. Hamilton, and Z. Su, "Has the bug really been fixed?" in *ICSE'10*, 2010, pp. 55–64.

[46] G. Robles and J. M. Gonzalez-Barahona, "Developer identification methods for integrated data from various sources," in *MSR'05*, 2005, pp. 1–5.

[47] K. Ehrlich and M. Cataldo, "All-for-one and one-for-all?: a multi-level analysis of communication patterns and individual performance in geographically distributed software development," in *CSCW'12*, 2012, pp. 945–954.

[48] C. Bird, D. Pattison, R. D'Souza, V. Filkov, and P. Devanbu, "Latent social structure in open source projects," in *FSE'08*, 2008, pp. 24–35.

[49] H. Keramati and S.-H. Mirian-Hosseinabadi, "Integrating software development security activities with agile methodologies," in *ICCSA'08*, 2008, pp. 749–754.

[50] E. Giger, M. Pinzger, and H. C. Gall, "Can we predict types of code changes? an empirical analysis," in *MSR'12*, 2012, pp. 217–226.

[51] S. Wang, C. Bansal, N. Nagappan, and A. A. Philip, "Leveraging change intents for characterizing and identifying large-review-effort changes," in *PROMISE'19*, 2019, pp. 46–55.

[52] A. D. Well and J. L. Myers, *Research design & statistical analysis*. Psychology Press, 2003.

[53] A. Agrawal, A. Rahman, R. Krishna, A. Sobran, and T. Menzies, "We don't need another hero?: the impact of heroes on software development," in *ICSE-SEIP'18*, 2018, pp. 245–253.

[54] S. Majumder, J. Chakraborty, A. Agrawal, and T. Menzies, "Why software projects need heroes (lessons learned from 1100+ projects)," *arXiv preprint arXiv:1904.09954*, 2019.

[55] S. Koch and G. Schneider, "Effort, co-operation and co-ordination in an open source software project: Gnome," *Information Systems Journal'02*, vol. 12, no. 1, pp. 27–42, 2002.

[56] S. Krishnamurthy, "Cave or community?: An empirical examination of 100 mature open source projects," 2002.

[57] G. Robles, J. M. Gonzalez-Barahona, and I. Herraiz, "Evolution of the core team of developers in libre software projects," in *MSR'09*, 2009, pp. 167–170.

[58] S. Kim, H. Zhang, R. Wu, and L. Gong, "Dealing with noise in defect prediction," in *ICSE'11*, 2011, pp. 481–490.

[59] M. S. Zanetti, I. Scholtes, C. J. Tessone, and F. Schweitzer, "The rise and fall of a central contributor: dynamics of social organization and performance in the gentoo community," in *CHASE'13*, 2013, pp. 49–56.

[60] H. Jiang, J. Zhang, H. Ma, N. Nazar, and Z. Ren, "Mining authorship characteristics in bug repositories," *SCIS'17*, vol. 60, no. 1, 2017.

[61] M. Zhou and A. Mockus, "Who will stay in the floss community? modeling participant's initial behavior," *TSE'15*, vol. 41, no. 1, pp. 82–99, 2015.

[62] ——, "What make long term contributors: Willingness and opportunity in oss community," in *ICSE'12*, 2012, pp. 518–528.

[63] M. Gharehyazie, D. Posnett, B. Vasilescu, and V. Filkov, "Developer initiation and social interactions in oss: A case study of the apache software foundation," *EMSE'15*, vol. 20, no. 5, pp. 1318–1353, 2015.

[64] L. López-Fernández, G. Robles, J. M. Gonzalez-Barahona, and I. Herraiz, "Applying social network analysis techniques to community-driven libre software projects," *IJITWE'06*, vol. 1, no. 3, pp. 27–48, 2006.

[65] L. Lopez-Fernandez, G. Robles, J. M. Gonzalez-Barahona *et al.*, "Applying social network analysis to the information in cvs repositories," in *MSR'04*, 2004, p. 101–105.

[66] S. L. Toral, M. d. R. Martínez-Torres, and F. Barrero, "Analysis of virtual communities supporting oss projects using social network analysis," *IST'10*, vol. 52, no. 3, 2010.

[67] T. Wolf, A. Schroter, D. Damian, and T. Nguyen, "Predicting build failures using social network analysis on developer communication," in *ICSE'09*, 2009, pp. 1–11.

[68] W. Zhang, L. Nie, H. Jiang, Z. Chen, and J. Liu, "Developer social networks in software engineering: construction, analysis, and applications," *SCIS'14*, vol. 57, no. 12, 2014.

[69] D. Surian, D. Lo, and E.-P. Lim, "Mining collaboration patterns from a large developer network," in *WCRE'10*, 2010, pp. 269–273.

[70] W. Zhang, S. Wang, Y. Yang, and Q. Wang, "Heterogeneous network analysis of developer contribution in bug repositories," in *CSC'13*, 2013, pp. 98–105.

[71] W. Bu, M. Xue, L. Xu, Y. Zhou, Z. Tang, and T. Xie, "When program analysis meets mobile security: an industrial study of misusing android internet sockets," in *FSE'17*, 2017, pp. 842–847.

[72] N. Munaiah, "Assisted discovery of software vulnerabilities," in *ICSE'18*, 2018, pp. 464–467.

[73] F. Camilo, A. Meneely, and M. Nagappan, "Do bugs foreshadow vulnerabilities?: a study of the chromium project," in *MSR'15*, 2015, pp. 269–279.

[74] A. Decan, T. Mens, and E. Constantinou, "On the impact of security vulnerabilities in the npm package dependency network," in *MSR'18*, 2018, pp. 181–191.

[75] M. Shahzad, M. Z. Shafiq, and A. X. Liu, "A large scale exploratory analysis of software vulnerability life cycles," in *ICSE'12*, 2012, pp. 771–781.

[76] D. Mu, A. Cuevas, L. Yang, H. Hu, X. Xing, B. Mao, and G. Wang, "Understanding the reproducibility of crowd-reported security vulnerabilities," in *USENIX Security'18)*, 2018, pp. 919–936.

[77] A. Ozment and S. E. Schechter, "Milk or wine: does software security improve with age?" in *USENIX Security Symposium'06*, 2006.

[78] N. Medeiros, N. Ivaki, P. Costa, and M. Vieira, "Software metrics as indicators of security vulnerabilities," in *ISSRE'17*, 2017, pp. 216–227.

[79] Y. Yang, J. Ai, X. Li, and W. E. Wong, "Mhcp model for quality evaluation for software structure based on software complex network," in *ISSRE'16*, 2016, pp. 298–308.

[80] Z. Xu, B. Chen, M. Chandramohan, Y. Liu, and F. Song, "Spain: security patch analysis for binaries towards understanding the pain and pills," in *ICSE'17*, 2017, pp. 462–472.

[81] S. Frei, M. May, U. Fiedler, and B. Plattner, "Large-scale vulnerability analysis," in *SIGCOMM'06*, 2006, pp. 131–138.

[82] J. Walden, J. Stuckman, and R. Scandariato, "Predicting vulnerable components: Software metrics vs text mining," in *ISSRE'14*, 2014, pp. 23–33.

[83] F. Rahman and P. Devanbu, "Ownership, experience and defects: a fine-grained study of authorship," in *ICSE'11*, 2011, pp. 491–500.

[84] B. Zhou, I. Neamtiu, and R. Gupta, "A cross-platform analysis of bugs and bug-fixing in open source projects: Desktop vs. android vs. ios," in *EASE'15*, 2015, p. 7.

[85] D. M. German, "The gnome project: a case study of open source, global software development," *Software Process: Improvement and Practice*, vol. 8, no. 4, pp. 201–215, 2003.