Automatic Commit Range Identification of Untagged Version

Yan Zhu, Lingfeng Bao, Chengjie Chen, Lexiao Zhang, Xin Yin and Chao Ni*
State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, China.
Email: {yan.zhu,lingfengbao,cchengjie,lexiaozhang,xyin,chaoni}@zju.edu.cn

***Abstract*—Aligning software product versions to commits is extremely important for fixing vulnerabilities in released versions. Existing work is proposed based on tags in the code repository. However, in practice, many software versions widely used in IT companies are reported with many high-risk vulnerabilities. In contrast, they have no indicator information (i.e., tags) in their source code repository. Such a situation results in the difficulty of tracing special versions to their particular commits for effectively fixing vulnerabilities.**

In this paper, we first study the software released on the Maven repository and hosted on GitHub. We collect and analyze the statistics of those versions that are reported with high-risk vulnerabilities but have no explicit information to locate the commit where they are released. To effectively locate the commits where a special version is released, we propose a novel approach named ContAlign and make a comprehensive comparison with three baselines that are proposed based on the two most common strategies: time-based ones and range-based ones. The experimental results on our built dataset indicate that ContAlign can obtain a good performance of 0.89 in terms of accuracy when identifying the commit range which covers the truth release commit of a specific version and improves baselines by 50.3%-102.2%. Besides, we also conduct a human study with 10 participants to evaluate the performance and usefulness of ContAlign, the user feedback indicates that ContAlign can effectively help participants align vulnerability versions to commits to the code repository.

***Index Terms*—Software Releases, Content Alignment, Git Tag, Maven**

I. Introduction

Release engineering generally integrates source code independently developed by contributors into a coherent software product containing both libraries and other resources required for software deployment and execution [1]. A release groups a few independent code changes into a deliverable version for specific stakeholders and is generally hosted on a remote repository for public availability (e.g., Maven Repository). Meanwhile, their corresponding source code is hosted on version control systems (e.g., Git) and is continuously evolving with fine-grained modifications on software artifacts.

Aligning software product releases/versions[1] to the source code repository is extremely important and has attracted much attention in both academic and industry [2]–[7], especially for untagged version during the process of software maintenance. For example, when fixing bugs/vulnerabilities, developers can significantly narrow the searchable code base by identifying the commits of the release that inserted the bug.

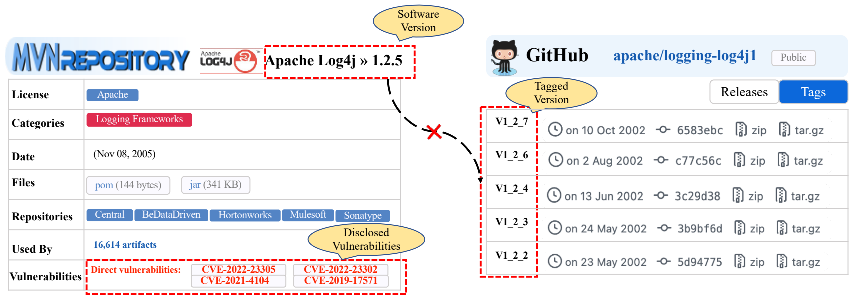
[1]Version and release are interchangeable used in this paper.

However, aligning released version without tags to commits is a non-trivial task. Although, tags are frequently used to indicate a specific commit of a release, version control systems (e.g., Git) have no built-in functionality to obtain all related commits of a given release. That is, those commits have the same content with the ones in released version. Hence, developers proposed to identify the commits that belong to a special released version based on two widely used strategies: ① *time-based ones* [3], [8]–[10] and ② *range-based ones* [2], [11], [12]. The time-based strategy assumes that any reachable commits in a specific time interval belong to a release, while the range-based strategy assumes that commits in the change path between two tags belong to the release. Both time-based strategy and range-based strategy are only implemented on the basis of tags in code repositories. For releases with no tags, they cannot work.

Unfortunately, in practice, many software versions widely used in IT companies are reported with several high-risk vulnerabilities (i.e., CVSS$>$7)[2], while they have no tags in their corresponding code repository. For example, we find that 553 releases with high-risk but with no tags in code repository are reported in Maven repository in recent 10 years. Such a situation leads to the difficulty of tracing a special version to their particular commits in effectively fixing vulnerabilities tasks.

In this paper, we focus on fixing high-risk vulnerabilities on untagged versions. We first study the software released on the Maven repository and hosted on GitHub. We collect and analyze the statistics of those versions which are reported with several high-risk vulnerabilities but have no explicit information (i.e., tags) to locate the commits where they are released. To effectively and efficiently locate the commits where a special version is released, we then propose a novel approach named ContAlign with two main phases: ❶ pre-processing phase for filtering candidate commits, and ❷ content mapping phase for content comparison between two files. We make a comprehensive comparison with three baselines that are proposed based on the most two common strategies: time-based ones and range-based ones. The experimental results on our built dataset indicate that ContAlign can obtain a good performance of 0.89 on identifying the commit range, which covers the truth released commit of a specific version and improves baselines by 50.3%-102.2%. Besides, we also conduct a

[2]https://nvd.nist.gov/vuln-metrics/cvss



License	Apache
Categories	Logging Frameworks
Date	(Nov 08, 2005)
Files	pom (144 bytes) jar (341 KB)
Repositories	Central BeDataDriven Hortonworks Mulesoft Sonatype
Used By	16,614 artifacts
Vulnerabilities	Direct vulnerabilities: CVE-2022-23305 CVE-2021-4104 CVE-2022-23302 CVE-2019-17571

V1_2_7	on 10 Oct 2002	6583ebc	zip	tar.gz
V1_2_6	on 2 Aug 2002	c77c56c	zip	tar.gz
V1_2_4	on 13 Jun 2002	3c29d38	zip	tar.gz
V1_2_3	on 24 May 2002	3b9bf6d	zip	tar.gz
V1_2_2	on 23 May 2002	5d94775	zip	tar.gz

Fig. 1: A motivating example of untagged version with vulnerabilities in code repository from Log4j V1.2.5.

human study with 10 participants to evaluate the performance and usefulness of ContAlign, the user feedback indicates that ContAlign can help participants align vulnerability version to untagged commits in the code repository.

The main contributions of this work are listed as follows:

- **Technique**. We propose a novel approach named ContAlign to effectively identify the minimum commit range for releasing a specific version. To the best of our knowledge, this is the first work to address such a practical scenario with a feasible solution.
- **Dataset**. We semi-automatically collect the mapping between software versions on Maven and GitHub repositories and analyze the statistics of versions with high-risk vulnerabilities. Totally, 2,892 versions from 37 Java projects are reported with high-risk vulnerabilities and with no tags.

II. Motivating Example

Vulnerabilities are inevitable in computer software and hardware. Once disclosed, vulnerabilities should be systematically managed and documented, which is critical for the practitioners and researchers to develop patches and prevention mechanisms [13]–[15]. Common Vulnerabilities and Exposures (CVE)[3] is the most influential vulnerability database and has recorded 240,830 vulnerability entries by July 10, 2024. Each CVE entry has an identification number (e.g., CVE-2024-6677) as well as the must-provided vulnerability information, which describes several aspects of the vulnerability, including vulnerability type, vulnerable component in which product and version(s), attacker type, impact, and attack vector. In which, vulnerable component in which product and its corresponding version can help participants quickly identify the location of vulnerabilities. Therefore, accurately obtaining the relationship between the software product version and its corresponding source code in the repository does a great favor for developers to fix the vulnerability.

Though most software products (e.g., open source products) are released with identified versions and well been matched with the tags managed in source code repository (e.g., GitHub), there still has some products missing a few accurate matching information between product versions and their tags in code repository.

[3]https://cve.mitre.org/index.html

For example, Log4j is an extremely popular software product from Apache community, which aims at providing professional logging functionality with friendly usage. However, as shown in Fig. 1, we find that Log4j has been disclosed four particular vulnerabilities in V1.2.5 in 2022. That is, developers may need to check out their code repository back to the commit where it is tagged as 1.2.5. However, though developers from Log4j in GitHub have managed their versions well with corresponding tags in most cases, developers forget to tag the commit when they released Log4j from V1.2.5, which will highly increase the difficulty to developers in locating and fixing the corresponding vulnerabilities in source code repositories. Meanwhile, we find that this situation is a non-trivial one (2,892 versions from 37 projects reported with high-risk vulnerabilities), and consequently motivates us to propose an approach to effectively locate the commits where a specific software product version is released.

III. Dataset

In this section, we describe the process of the dataset construction, including selecting software products in Maven, identifying code repositories, and building the traceability linking Maven software versions to commits. Then, we present some statistics of our dataset.

A. Dataset Construction

In our study, we focus on building the traceability between software versions with vulnerabilities and commits in code repositories. Therefore, in this section we investigate how many software versions with vulnerabilities can be linked to commits automatically using an easy approach (e.g., based on tag names in code repositories). These software versions can also be used as a ground truth to evaluate our approach (see Section IV). Additionally, we can also know how many links between software versions with vulnerabilities and commits are missing. The process of dataset construction is as follows:

Select software products in Maven. We get a list of OSS software products published in the Maven repository from a big IT company. There are 6,015 software products with 483,234 versions in this list. These software products in the list are widely used by the internal and commercial software products of this company, such as Log4J, spring-frameworks, etc. Then, we identify the software versions with vulnerabilities by crawling their web pages (see Fig 1). Out of 483,234 versions, 3.17% (15,308) software versions are associated with at least one vulnerability.

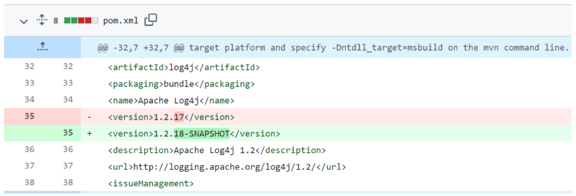
TABLE I: Statistics of Maven Software Versions

	Total	Tagged	Untagged
All	483,234	426,228	57,006
Vulnerabilities	15,308	12,393	2,915

Identify code repositories. Since the software products in Maven do not have the code repository information, we need to link them and their corresponding code repositories. As most OSS software is hosted on GitHub, we focus on GitHub in this study. We use the following steps to map a Maven software product to its GitHub repository: First, we map the software products in Maven and GitHub repositories based on the text similarity of their names automatically. Then, we manually check whether the mapping is correct based on the information provided by Maven and GitHub. If an incorrect mapping is found, we try to find the repository using the GitHub search and Google search. Finally, these 6,015 software products in Maven are linked to 2,702 GitHub repositories. Some software products in Maven can be mapped to the same GitHub repository. For example, the project *Netfilx/conductor*[4] on GitHub releases multiple components in Maven, such as *conductor-core* and *conductor-common*.

Build the traceability linking Maven software versions to commits. Next, we want to link the versions of software products in Maven to the commits of the corresponding code repositories. Also, we can know how many versions of software products in Maven cannot be mapped to the commits in code repositories. For each software product in Maven, we extract its version list. We also extract all the tags from its corresponding code repository since a tag is usually assigned to a commit for a release version by project maintainers in the Git workflow, e.g., *0.2.0-RC0* in the project Apache/Hadoop.

Then, we map the software versions in Maven to the tags in the code repository. The format of Maven software versions is *x.x.x* (e.g., Apache Log4j 1.2.5 in Fig 1). On the other hand, the tag name in the code repository varies across projects and developers, for example, *v_1_2_7* in Apache/Log4j1 and *rel/2.0* in logging-log4j2. Therefore, we use several regular expressions to extract version numbers from these tag names, e.g., (\d+).(\d+).(\d+).



```
8 pom.xml
@@ -32,7 +32,7 @@ target platform and specify -Dntdll_target=msbuild on the mvn command line.
      <artifactId>log4j</artifactId>
      <packaging>bundle</packaging>
      <name>Apache Log4j</name>
-     <version>1.2.17</version>
+     <version>1.2.18-SNAPSHOT</version>
      <description>Apache Log4j 1.2</description>
      <url>http://logging.apache.org/log4j/1.2/</url>
      <issueManagement>
```

Fig. 2: log4j1 pom.xml

The majority of software versions in Maven can be linked to the commits in the code repositories based on tags. However, we fail to identify tags in code repositories for a small number of software versions in Maven. For the remaining software versions, we find another way to identify additional links between software versions in Maven and commits in code repositories, that is, by mining the commits in which the version information is changed in the configuration files (i.e., *pom.xml* in Maven projects). The reason is that the version information in the configuration file is always modified when releasing a new version of software product. For example, in a commit of the project *Apache/Log4j1*[5], we find that the version is changed from *1.2.17* to *1.2.18-SNAPSHOT* (see Fig. 2). Thus, we believe that this commit can be linked to the version *1.2.18*. However, we do not identify the tag for this version.

For each software product, we first locate its *pom.xml* file in its corresponding code repository. This is because a code repository can have multiple submodules for different software products. Then, we use the `git log --- submodule/pom.xml` command to extract all commits that are related to modifications on the configuration file. Finally, we check whether the version information is changed in a commit. If so, we extract the version information and think this version is linked to the commit.

B. DATASET DESCRIPTION

Table I presents some statistics of Maven software versions in our dataset. As shown in the table, a lot of links between Maven software versions and commits in code repositories are missing. 11.80% (57,006/483,234) software versions cannot be linked to a commit in the code repositories. For software versions with vulnerabilities, more untagged versions (19.04% $\approx 2,915/15,308$) are identified. Thus, we believe that it is necessary to build the traceability between software versions and commits.

Furthermore, out of 2,915 untagged versions, there are 674 versions that is associated with high-risk vulnerabilities (i.e., the severity score of an CVE vulnerability is larger than 7). These 674 untagged versions with high-risk vulnerabilities are from 37 Maven software. Table II provides the statistics of these 37 Maven software that contain versions with high-risk vulnerabilities. Due to the page limitation, we only list top 10 softwares with most versions of high-risk vulnerabilities. There are 2,892 software versions that are linked to a commit in the code repository. Notice that we link 44 additional versions to a commit by mining the commits in which version information is changed. These 37 Maven softwares belong to different categories, e.g., *spring-core* is a J2EE framework and *fastjson* is a json library.

IV. APPROACH

Accurately identifying a specific commit of a released version is extremely difficult, especially for untagged version, we therefore design an algorithm named ContAlign to map a software product version to its corresponding commits in code repository. In particular, ContAlign has two main phases: ❶

[4]https://github.com/Netflix/conductor

[5]https://github.com/apache/logging-log4j1/commit/0311ea67

TABLE II: Maven software that contains versions with high-risk vulnerabilities

Maven Software	# Tag			# Untagged			GitHub Repo	# Commit
	All	Vuln	High-Risk Vuln	All	Vuln	High-Risk Vuln		
h2	20	20	20	108	108	108	h2database/h2database	14,017
nexus-core	80	50	50	87	85	85	sonatype/nexus-public	3,092
jackson-mapper-asl	4	4	4	78	76	76	codehaus/jackson	2,404
mpxj	70	52	45	55	55	55	joniles/mpxj	2916
fastjson	122	77	77	98	52	52	alibaba/fastjson	7,173
spring-core	162	120	120	48	46	46	spring-projects/spring-framework	29,087
tomcat-embed-core	174	166	166	79	41	41	apache/tomcat	68,502
hutool-all	60	0	0	85	23	23	dromara/hutool	5,837
nexus-extdirect	64	34	34	17	15	15	sonatype/nexus-public	3,092
jackson-dataformat-xml	109	60	60	14	13	13	FasterXML/jackson-dataformat-xml	1,862
. . .								
Total	2,892	2,078	1,893	994	749	674		377,071

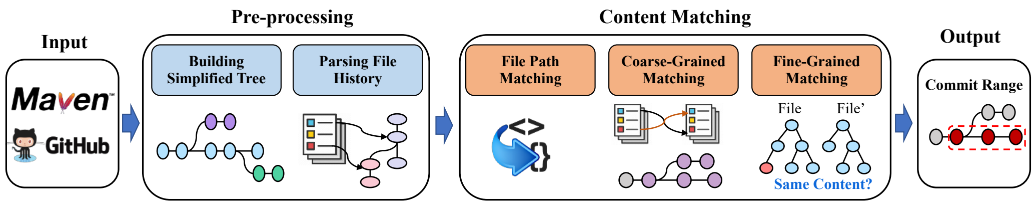


Fig. 3: The framework of our approach

pre-processing phase and ❷ **content mapping phase**, takes a software and its identified code repository as input, and outputs a commit range. In such a range, each source code file in software version has the same content with the same source code file in code repository in each commit. ContAlign works through all branches iteratively in a specific code repository and the framework of ContAlign is illustrated in Fig. 3. We introduce them in detail as follows.

A. *Pre-processing Phase*

This phase does several preparations for mapping specific software version to corresponding commits, involving two main steps: (1) building simplified commit history tree; and (2) parsing file history.

(1) Building Simplified Commit History Tree. Software code repository intrinsically records all commit histories with the functionality of source control management system (e.g., Git considered in our study only). In practice, projects usually are developed in parallel in different branches for various purposes. A commit usually has its parent(s) (except the initialized commit) and its child(children) (except the latest commit), therefore all commits are connected logically and stored in the form of tree-structure in Git. Though Git have scientifically organized all commits, it builds a large data structure of the whole commits in a repository. In our study, for simplification and efficiency, we build a simplified commit history tree to precisely retrieve commit information in each branch. That is, each node only stores three kinds of information: commit index, commit parents, and submitted date. Such a simplified structure will be convenient for subsequent analysis tasks (e.g., Coarse-grained matching).

(2) Parsing File History. This step aims at parsing the adding, removing and renaming history of all files in source code repository. To represent relationship among all commits in chronological order, we set each commit with an incremental index of integer. For example, $commit_i$ means the commit is submitted to code repository ahead of the $commit_{i+1}$ in chronological order. Besides, each file may has several adding or removing history, for a specific file F, we denote them as $F_{add\_commit} = [add_1, add_2, \cdots, add_x]$ and $F_{remove\_commit} = [rmv_1, rmv_2, \cdots, rmv_y]$, where $0 \leq y$ & $0 < x$, both x and y represent the numbers of modifications to a specific file. Renaming operation is used to indicate whether two files are the same one, and is denoted as $\{apath : fpath_1, bpath : fpath_2\}$, where $apath$ and $bpath$ represent the path of a file before and after renaming operation, respectively. Therefore, we add the commits of the renaming operation into F_{remove_commit} with $fpath_1$, and into F_{add_commit} with $fpath_2$. Right here, we have all files history involving adding, removing and renaming operations.

B. *Content Matching Phase*

Content matching phase aims at identifying the minimal commit range in source code repository where all source code files in each commit are 100% same with the source code file in a specific software product version in the view of content. In particular, it contains three main steps: (1) file path matching, (2) coarse-grained matching, and (3) fine-grained matching.

(1) File Path Matching. File matching refers to matching files in the software released package to files in the source code repository. In practice, the structure of files' path in source code repository may have a little difference with the structure in software released package. That is, the released software product versions typically remove the prefix of file paths in code repositories. For example, as shown in Fig. 4, we can see the structure difference in paths of files in project *undertow-core v1.4.9*. Therefore, we have to address the influence of structure difference in path to accurately identify the relationship between the files in software released package and the files in source code repository.

We propose a metric named *Path Similarity Score* (PCS) to calculate the similarity between two file paths. Firstly, we obtain all files with their paths submitted to source code repository, denoted as $rdirs$, and extract all files with their paths in software released package, denoted as $sdirs$. Then, we split each file's full path ($rdir$ and $sdir$) into a list of tokens by slashes, denoted as $rdir_l$ and $sdir_l$. For two paths having the same filename (i.e., last tokens in both $rdir_l$ and $sdir_l$ are same), we search the first token of longer token list which firstly matches the first token in the shorter token list. Then, we extract the sub-list from the longer token list which matches the first token and the last token in the shorter token list, denoted as $cdir$. Moreover, we define the average length of $cdir$ and $sdir$ as N, and the numbers of same tokens in both $cdir$ and $sdir$ is n. Formally, $N = \frac{|cdir|+|sdir|}{2}$.

We also calculate the longest common sub-sequence length $LLCS$ between $cdir$ and $sdir$ with the consideration of the impact of paths' order on matching files. Finally, the path similarity score PSC can be defined as follows: $PSC = \frac{n}{N} \times \frac{LLCS}{N}$.

For example, one of the files in the project *undertow-core* source code package is "io/undertow/security/Account.java", and another similar file in the code repository is "core/src/main/java/io/undertow/security/Account.java". Then, the file names are matched (e.g., "Account.java"). After splitting the two paths by "/", $sdir_l$ equals to ["io","undertow","security","Account.java"], and $rdir_l$ equals to ["core","src","main","java","io","undertow","security", "Account.java"]. We obtain the first same token "io" of $sdir_1$, then extract the sub-sequence from $sdir_l$ into $cdir_l$, that is, ["io","undertow","security","Account.java"]. Both N and n equals to 4, and $LLCS$ also equals to 4, therefore PSC equals 1. Notice that N may be no less than 2 since two tokens must be the same ones (i.e., first token and last file name). ContAlign matches the changed files in the code repository for each source file in software release package, calculates the path similarity score between paired two files, and selects the one or more with the largest *PSC* value for each file in software release pack as candidates.

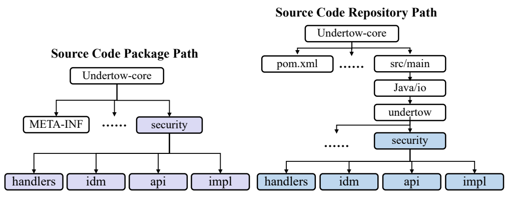


Fig. 4: File path structure comparison in untertow-core between software release version and source code repository.

(2) Coarse-grained Matching. Prior step builds the matching relationship between files in software release version and files in source code repository. Then, we build a file commit history tree tracing from the commit it is removed to the commit it is added. For those files which have ever not been removed, we use the latest commit to simulate that it is removed. Therefore, each file separately has a set of commit history and we finally obtain the intersected commit history tree of all files (denoted as $Set_{intersected\_commit}$). Right here, we ensure that files in both software release version and source code repository have the same file path in each commit of $Set_{intersected\_commit}$.

(3) Fine-grained Matching. Coarse-grained matching can only ensure the consistence of file path in both software release version and source code repository. In this step, we aim at the consistence of files' contents. To achieve this goal, we separately build two parsing trees for each file with same full paths. We use parsing tree instead of abstract syntax tree (AST) with the consideration that two ASTs may have sample structures but have different contents. For example, we find that project *undertow-core-1.4.9* totally has 740 files, among which 130 files have same ASTs structure with ones in code repository but have different content. As for the two parsing tree, one represents the files' content in software release version (denoted as $Tree_{version}$), while another one represents the files' content in source code repository (denoted as $Tree_{repo}$). We utilize ANTLR[6] tool as the parsing tree builder, which is a powerful parser generator for reading, processing, executing or translating structured text or binary files. Then, we iterate each commit in $Set_{intersected\_commit}$ and check whether each file with same paths has the same parsing tree (i.e.g, $Tree_{version}$ equals to $Tree_{repo}$). For all files in a commit have same parsing trees, the corresponding commit will be saved as our target commit, which may release the corresponding release of such a software. Notice that, ContAlign may returns a range of commit and in each commit all source code files do have the same content with the source code files in software release version. The differences in these commits may lie in the modifications involving documenting, testing and removing space.

V. Evaluation

To evaluate ContAlign's effectiveness and usability, we conduct several quantitative and qualitative experiments. The section introduces the researcher question, the performance measures, and the experiment setting sequentially.

A. Research Questions

- **[RQ-1]: How well does the ContAlign perform in identifying the commit range of an untagged software product version?**
  The main object of our study is to accurately locate the commit (the commit range) for a released software product version reported with vulnerabilities. Therefore, we are concerned about the ContAlign's effectiveness. To evaluate it, we first build a ground truth from hundreds of open source Java projects reported with high risk vulnerabilities (i.e., *CVSS* > 7)[7] in the Maven repository. After that, we make

[6]https://www.antlr.org/
[7]https://nvd.nist.gov/vuln-metrics/cvss

some comparisons between ContAlign and some baseline approaches to quantitatively show ContAlign's effectiveness.

- **[RQ-2]: Is ContAlign useful for developers to locate the commit to develop patches for identified vulnerabilities?** The final goal of our study is to check whether ContAlign can work well in practice. Therefore, we study to analyze its usefulness to developers by conducting a user-study survey through a questionnaire.

B. Quantitative Experiments Metrics

To evaluate the effectiveness of ContAlign, we adopt the following two performance metrics (i.e., Accuracy and MAE) to measure how well our approach can identify the commit(commit range) where a specific software product version is released.

Before we define the two performance metrics formally, we have to introduce several settings in our performance calculation. Notice that, in practice, developers may work together on different branches for various characteristics as shown in Fig. 5(a), meanwhile these commits are recorded by source control management system (e.g., GIT in our study) according to their chronological order as shown in Fig. 5(b). However, our approach usually identifies a range of commits where all source code files have the same content with the ones in a specific software product version as shown in Fig. 5(c). Therefore, to easily evaluate the effectiveness of ContAlign and to represent relationship among all commits in chronological order, we provide each commit with an incremental value of integer. For example, $Commit_i$ means the commit is submitted to code repository ahead of the $Commit_{i+1}$ in chronological order and meanwhile the two commits are adjacent ones. Assuming the ground truth of an untagged version was released at $Commit_i$, and our approach gives out a candidate commit range is $[Commit_x, Commit_y]$, $i, x, y \in [0, max]$. In which, $Commit_0$ represents the initialization commit, while the $Commit_{max}$ represents the latest commit in the code repository. Based on above introduction, we give the two performance metrics as follows.

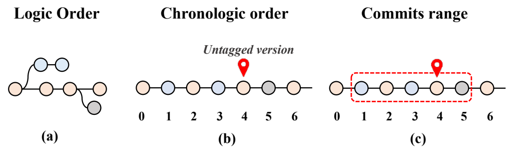


Fig. 5: An example to illustrate the organization of commits.

Accuracy. It is a correct identification of commit range (i.e., $[Commit_x, Commit_y]$) for a untagged software product version if and only if the commit range includes the actual ground truth commit (i.e., $Commit_i$). That is, $Commit_i \in [Commit_x, Commit_y]$. Therefore, we define the metric *Accuracy* to measure the proportion of the numbers of hit commit ranges over all tested versions. Formally,

$$Accuracy = \frac{1}{N}\sum_{i=1}^{N}\begin{cases}1, & Commit_i \in [Commit_x, Commit_y]\\ 0, & Commit_i \notin [Commit_x, Commit_y]\end{cases}$$

where N is the number of total test cases, i, x, y is the index of commit. The higher value the $Accuracy$ has, the better performance ContAlign obtains.

MAE. To evaluate the stability of ContAlign's performance when it correctly, we adopt mean absolute error (MAE) performance metric. MAE measures the amount of error in a statistical model, which assesses the absolute difference between the observed value and predicted values. The MAE equals zero if an approach has no error. As the error increases, its value increases. Notice that, in our study, our approach actually provides a commit range rather a specific commit index. Therefore, we calculate MAE with a few modifications as follows, $MAE = \frac{1}{2N}\sum_{1}^{N}|x - i| + |y - i|$, where i is the index of ground truth commit, x, y are the corresponding predicted indexes of commit range, and N is the number of total test cases. The lower value the MAE has, the better performance ContAlign has.

VI. Results

A. RQ-1: How well does the ContAlign perform in identifying the commit range of an untagged software product version?

Experiment Design. To evaluate the performance of ContAlign, we firstly manually build a large-scale dataset which contains 6,015 software products with 483,234 versions widely used in IT companies, and 2,702 corresponding code repositories hosted on GitHub. The detailed introduction to the dataset can be referred to Section III. Since ContAlign does require a few running time and the scale of our dataset is extremely large, we then pay attention to software products whose versions are reported with high risk vulnerabilities (i.e., *CVSS* $>$ *7*). We obtain 37 projects with 2,892 versions, and sample 10% versions from each project by filtering out versions whose commits cannot be found in their corresponding GitHub repositories. Notice that, we keep the same distribution of our sampled dataset with original 37 projects. Finally, 271 software versions are used to evaluate ContAlign's performance.

Furthermore, to comprehensively demonstrate the effectiveness of ContAlign, we also consider a few state-of-the-art baselines. In the scenario of assigning commits to a release, the most common strategies are ❶ **time-based strategy** and ❷ **range-based strategy** [16]. The time-based strategy assumes that any reachable commits in a specific time interval belongs to a release. As for the range-based strategy, it assigns the commits in the change path between two tags to the release. Therefore, inspired by two strategies, we comprehensively compare the accuracy of our approach with the following three baselines: (1) *Range*, (2) *RangeGuess* and (3) *DayRange*. Before introducing the three baselines, assuming we have a specific software product version X released on Maven repository on the date of *T*, and it is tagged at commit $Commit_{gt}$ in code repository in branch *B*. Besides, $Commit_{gt-}$ and $Commit_{gt+}$ represents the ground truth commits of the two adjacent versions: X_{prior} represents the version prior to X, while X_{after} represents the version after X. The three baselines can be described as follows.

Range means the largest commit range of a specific untagged version X in code repository in most cases since their adjacent versions may exist in different branches. We remove such versions that their adjacent version are tagged in different branches since such a situation accounts for only a small proportion in our study (i.e., 20.1%). Therefore, we use the commit (i.e., $Commit_{gt-}$) of prior version to version X as the lower bound of range, while use the commit (i.e., $Commit_{gt+}$) of later version to version X as the upper bound of range. Formally, *Range* is defined as $[Commit_{gt-}, Commit_{gt+}]$.

RangeGuess means randomly sampling two integers (i.e., *(pv, lt)*) from the range of $[Commit_{gt-},\ Commit_{gt+}]$ for the ground truth of a specific software product version X, in which $0 \leq gt- < gt < gt+$ and $0 \leq gt- \leq pv \leq lv \leq gt+$. Formally, *RangeGuess* is defined as $[Commit_{pv}, Commit_{lv}]$. Notice that, to some degree, *RangeGuess* is similar to our approach ContAlign.

DayRange means that it represents the first commit (denoted as $Commit_{fst}$) and the last commit (denoted as $Commit_{lst}$), which are submitted to code repository on the date of *T*. Formally, *DayRange* is defined as $[Commit_{fst},\ Commit_{lst}]$.

Notice that, *Range* always covers the ground truth of version X, while both *RangeGuess* and *DayRange* may sometimes miss the ground truth commit. Moreover, considering the randomness in *RangeGuess*, we also execute this method for 10, 30, 50 times and report the average performance for each setting.

Results. The evaluation results are shown in Table. III$\sim$Table. V. The best performances are highlighted and the improvement values indicate the difference between ContAlign and baselines.

Table. III shows the average performance of all approaches on identifying the correctness of the commit range, which covers the ground truth commit. As mentioned before, *Range* will always cover the ground truth, therefore, the accuracy of *Range* is 100%. Therefore, based on the results, we find that ContAlign achieves the best performance except for *Range*. In particular, ContAlign obtains a good performance of 0.89 in terms of accuracy, which means that ContAlign can precisely locate the range of truth commit and reduce false alarms. Besides, ContAlign improves RangeGuess and DayRange by 96.4%-102.2% and 50.3%, respectively. The poor results of both *RangeGuess* and *DayRange* demonstrate that aligning the version to commit in the code repository is not easy and cannot be addressed well by simple strategy-based approaches.

TABLE III: Performance comparison between ContAlign and baselines.

Metric	Range	RangeGuess			DayRange	ContAlign
		10	30	50		
Accuracy	1.0	0.44	0.46	0.45	0.59	**0.89**
Improv.	/	102.2%	96.0%	96.4%	50.3%	

To compare the conciseness of all approaches, we also make a comparison between ContAlign and baselines on those versions where two compared approaches correctly identify a commit range covering the ground truth commit. Table. IV shows the comparison results in terms of MAE on average. The columns "# Hit" means the numbers of software versions, which are correctly identified by their commit range. According to the results, we achieve the following observations:

(1) ContAlign obtains 4.62, 4.53, 4.45, 4.68, which improves *Range*, *RangeGuess-10*, *RangeGuess-30*, and *RangeGuess-50* by 88.8%, 74.0%, 75.4%, and 73.9%, respectively.

(2) Though *Range* can always cover the ground truth commit in our setting, it also increases the number of unsuitable commits (files in such commits cannot satisfy the content for releasing the specific version) and consequently increases the workflow of the developer for checking.

(3) By analyzing the performance of *DayRange*, we find that *DayRange* can obtain the best performance of MAE if *DayRange* correctly identify the commit range. It seems that on average developers may not submit many commits and consequently decrease the error of MAE. Meanwhile, ContAlign achieves similar performance with *DayRange*, which indicates the effectiveness of our approach.

(4) Overall, ContAlign has a stable performance and can extremely decrease the false positive when identifying the commits of a released version.

TABLE IV: Performance comparison between ContAlign and baselines when two approaches correctly identify a commit range covering the ground truth commit.

Approach	MAE	# Hit	*Improv.*
Range	41.26	242/271	88.8%
ContAlign	**4.62**		
RangeGuess-10	17.41	110/271	74.0%
ContAlign	**4.53**		
RangeGuess-30	18.55	111/271	75.4%
ContAlign	**4.56**		
RangeGuess-50	17.93	111/271	73.9%
ContAlign	**4.68**		
DayRange	**3.98**	144/271	/
ContAlign	4.44		

Meanwhile, we also present the results of ContAlign on those versions where ContAlign correctly identifies the commit range covering the ground truth commit while baselines incorrectly identify the ones. According to the results shown in Table. V, we observe that ContAlign also performs well and effectively figures out the acceptable commit range.

TABLE V: Performance of ContAlign when only ContAlign correctly identifies a commit range covering the ground truth commit.

Baselines	ContAlign *v.s.*				
	Range	RangeGuess			DayRange
		10	30	50	
MAE	/	4.70	4.67	4.57	4.88
# Hit		131.3/271	130.44/271	130.86/271	98/271

✍ **RQ-1** *ContAlign can accurately and effectively identify the commit range covering the ground truth commit of the version without tags information in a code repository.*

TABLE VI: Questionnaire for checking both modification types and commit boundary

Question	Possible Answers
Q1: What kinds of operations occur in the commit range? How many times?	Ⓐ. Formatting/cosmetic changes (e.g., newline, white space, moving a bracket to the next line, etc.),___ Ⓑ. Modifications on comments,___ Ⓒ. Modifications on testing,___ Ⓓ. Modifications on non-java files (e.g., xml, txt,html, etc.),___ Ⓔ. Merge commit,___ Ⓕ. Modifications on file in unrelated directory, ___ Ⓖ. Other, ___
Q2: what is the difference between C_L and C_{L-1}?	❶. Same content of Java source code with released version in commit C_L, while they are not same in C_{L-1} ❷. No difference ❸. Other, ___
Q3: Can add C_{L-1} into commit range [C_L,C_R]?	❶. Yes ❷. No ❸. Not Sure
Q4: what is the difference between C_R and C_{R+1}?	❶. Same content of Java source code with released version in commit C_R, while they are not same in C_{R+1} ❷. No difference ❸. Other, ___
Q5: Can add C_{R+1} into commit range [C_L,C_R]?	❶. Yes ❷. No ❸. Not Sure

B. RQ-2: Is ContAlign useful for developers to locate the commit to develop patches for identified vulnerabilities?

Experiment Design. To evaluate the usefulness of ContAlign in practice, we manually analyze how well ContAlign perform on identifying the commit ranges from the view of size to the modification types.

Results. We conduct a user study including 10 participants and 10 randomly sampled versions from the 24 unique product versions. All participants have at least 3-years of programming experience with Java and they are not the authors of this paper. ContAlign generally has a range (denoted as [C_L, C_R]) of possible commits, in which, each source code file in Java has 100% content the same as the corresponding one in the released source version. Meanwhile, we do not have a truth commit since no valid enough information is obtained from the actual developer. Manually checking each file's content (e.g., *.html, *.css, *.md, *.xml, *.txt, etc) in the released source code version requires much labor (the number of files ranges from 20-621 in the sampled versions). Therefore, in this use study, we aim to figure out the kinds of operations within and without the obtained commit range with the following questionnaires. Notice that C_{L-1} represents the prior commit of C_L, while C_{R+1} represents the latter commit of C_R.

According to the results from user study, we obtain the following observations:

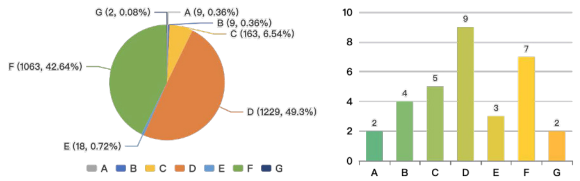


Fig. 6: Distribution of modification types in commit range.

(1) According to the results of Q2-Q5, we find that all participants agreed that both C_{L-1} and C_{R+1} cannot be added to commit range [C_L,C_R] generated by ContAlign in all studied cases, which means commit range [C_L,C_R] is the largest size and most possible commit range verified by ContAlign. The overall good performance (i.e., 10 projects with 145 generated commits involving 2,493 hunks) indicates that ContAlign can effectively narrow the searchable range of candidate commits.

(2) According to the results of Q1 shown in Fig. 6, we find that all code modifications that occurred in the commit range are unrelated to the source code of target components in the released version. In particular, modifications on non-java files (Answer D) and modifications on files in unrelated directories (Answer F) are the most common operation types, accounting for 49% (1,229/2,493) and 43%(1,063/2,493), respectively. The two types of operations types occur in 9 projects and 7 projects. Besides, one participant replied with another operation type (Answer G) with two 2 hunks in two projects. Such modifications add new source code files to other components. Eventually, the manually checked results confirm that to some degree, ContAlign can increase the efficiency of content comparing since the modifications in the commit range are unrelated to the source code of a specific release, and consequently increases the credibility of those commits to be the truth commit of the specific release.

✍ **RQ-2** *ContAlign can effectively narrow the searchable range of candidate commits for a released version and increase the efficiency of content checking between release and commit.*

VII. Threats to Validation

Threats to Internal Validity may come from two aspects: (1) code quality; (2) establishing ground truth. As for code quality, there may has potential mistakes in our implementation of our algorithm and other baselines. To minimize such threats, we not only implement these approaches by pair programming but also widely use high-quality third-party library (e.g., ANTLR). The authors also carefully double-check the experimental scripts to ensure their correctness. As for establishing ground truth, considering the complexity of locating the correct code repositories, we firstly use search techniques to filter out candidate ones, then we recruit developers from an IT company to manually check the relationship between software product version and its corresponding source code repository. Such a process consumes a lot of time and labor and may potentially introduce some mistakes. To address it, we only recruit developers with at least three-years developing experience in Java and those developers are familiar with mostly of those selected software products in Maven repositories.

Threats to External Validity may exist in our studied dataset. Though we have fully analyzed the situation of the software product version with untagged labels on the code repository, and the effectiveness of our approach has been verified on randomly sampled products, the studied dataset still has a such a limitation about the programming language used in studied projects, which are developed in Java programming language. Projects developed by other popular programming language (e.g., C/C++ and Python) have not been considered. Theoretically, our approach can be easily extended to any programming languages with a few modification of building the corresponding programming language parsing tree. Thus, more datasets should be collected and explored in future work.

Threats to Construct Validity mainly lie in the adopted performance metrics in our evaluations. To minimize such threat, we adopt a few widely used performance measures. In particular, we consider accuracy and MAE to evaluate our approach' effectiveness and efficiency.

VIII. Related Work

Software traceability is the ability of providing trace information on requirements, design, implementation, and maintenance of a system [17], which has been widely studied in the scenario of academic [18]–[22]. Bashir et al. [18] studied the traceability techniques on requirement management. They defined scope as a boundary for traceability and defined four types of coverage in traceability. Galvao et al. [19] conducted a survey of traceability approaches in model-driven engineering and provided an overview of the current state of traceability from both research and practice view. Jaber et al. [20] conducted a research on evaluating the benefits of tracing the information among various software engineering tasks (i.e., requirements, design, code, code inspections, builds, defects, and test artifacts) during the maintenance phase, and proposed a traceability link model categorizing different types of traceability links based on stakeholders' roles.

Tracing the relationship between the software product artifacts and the source code is a special task of software traceability, and is also an extremely in software maintenance. According to their different purposes, these study can be grouped into the following topics: (1) aligning commits to releases; (2) aligning commits to related commits; (3) aligning commits to issues; (4) others.

Aligning commits to releases. Shobe et al. [2] proposed an approach based on range-based strategy to establish the mapping from a particular release to the specific earliest and latest revisions. However, their approach can only analyze two adjacent versions with the information of manually analyzed logs and they also did not provided detailed experimental results about their approach. Khomh et al. [3], [23] conducted a study to compare the releases regarding quality. In prior work, they only check out the versions and compare them externally, which is similar to compare the last commits of the releases only [23]. Following that, they adopted the time-based strategy to extract more information including the developer of a commit, the number of commits, as well as their size [3]. Mäntylä et al. [10], [24] used time-based strategy to compare the releases regarding software testing. However, they only assigned the commits to Firefox's major releases and explained the difficulty in linking commits to specific releases. Clark et al. [25] compared the software product releases regarding the security of the produced code by only considering the last commit. Souza et al. [26], [27] used time-based strategy to analyze reversing commits on rapid releases. Recently, Pinto et al. [16] conducted a large-scale empirical study on 100 open source projects hosted at GitHub. They assessed two widely used strategies (i.e., time-based and range-based) for assigning commits to releases. Their results shown that generally range-based strategy outputs than times based strategy. But, some releases still were mis-classified by range-based strategy.

Aligning commits to related commits. To avoid the risk of missing commit dependencies in the process of selective code integration, Dhaliwal et al. [28] propose two approaches (i.e., a developer-guided approach and an automatic approach) to identify dependencies among commits on the basis of the dissimilarity levels learned from previous versions of a software products. The experimental results demonstrated their developer-guided approach can reduce integration failures by 94%, while the automated approach can reduce integration failure by 76%. Hammad et al. [4] also conducted a study to identify the related and similar commits from software repositories with an automatic approach.

Aligning commits to issues. Le et al. [5] proposed a new bug links approach *RCLinker* to address the issue that the link between issue reports and their corresponding commits are missing. Their approach fully utilized the rich contextual information and the summarizing techniques. The results on six projects demonstrated *RCLinker* outperforms *MLink* in terms of F-score by 138.66%. Sun et al. [6] proposed *FRLink* by considering non-source documents for both code and text feature comparison in order to increase the likelihood to recover missing links. Meanwhile, to reduce noise introduced by those unrelated files, *FRLink* identified source code files modified in commits. Their results indicated that *FRLink* improved *RCLinker* by 40.75% in terms of F-score.

Others. Kagdi et al. [29] proposed a heuristic approach based on frequent-pattern to uncover traceability patterns between source code files and other artifacts (e.g., documents) by using versions history, and their experimental results on a number of versions of the open source system KDE demonstrated its effectiveness. Abebe et al. [7] firstly analyzed and understand the nature of release notes. They found no consistent pattern for writing release notes and the information types listed in release notes varied from system to system. Therefore, they proposed a machine learning approach to automatically suggest important addressed issues to be listed in release notes.

Different from prior studies, our paper focus on aligning software product versions to their corresponding commits to find out the truth/suitable one for releasing such a version. Prior studied [16] are usually proposed on the basic of two strategies: times-based ones and range-based one. In our paper, we align software versions to commit by analyzing and comparing the content of related files in both versions and code repository, while increases the preciseness of the identification of candidate commits.

IX. Conclusion

To effectively align releases to commit, we propose a novel approach named ContAlign with two main phases: (1) preprocessing phase for filtering candidate commits, and (2) content mapping phase for content comparison between two files. We make a comprehensive comparison with three baselines that are proposed based on the most two common strategies: time-based ones and range-based ones. The experimental results on our built dataset indicate that ContAlign can obtain a good performance of 0.89 on identifying the commit range, which covers the truth released commit of a specific version and improves baselines by 50.3%-102.2%. Besides, we also conduct a human study with 10 participants to evaluate the performance and usefulness of ContAlign and these feedback indicates the usefulness.

Acknowledgements

This work was supported by the National Natural Science Foundation of China (Grant No.62202419), the Fundamental Research Funds for the Central Universities (No. 226-2022-00064), Zhejiang Provincial Natural Science Foundation of China (No. LY24F020008), the Ningbo Natural Science Foundation (No. 2022J184), and the State Street Zhejiang University Technology Center.

References

[1] B. Adams, S. Bellomo, C. Bird, B. Debić, F. Khomh, K. Moir, and J. O'Duinn, "Release engineering 3.0," *IEEE Software*, vol. 35, no. 2, pp. 22–25, 2018.

[2] J. F. Shobe, M. Y. Karim, M. B. Zanjani, and H. Kagdi, "On mapping releases to commits in open source systems," in *Proceedings of the 22nd International Conference on Program Comprehension*, 2014, pp. 68–71.

[3] F. Khomh, B. Adams, T. Dhaliwal, and Y. Zou, "Understanding the impact of rapid releases on software quality," *Empirical Software Engineering*, vol. 20, no. 2, pp. 336–373, 2015.

[4] M. Hammad, "Identifying related commits from software repositories," *International Journal of Computer Applications in Technology*, vol. 51, no. 3, pp. 212–218, 2015.

[5] T.-D. B. Le, M. Linares-Vásquez, D. Lo, and D. Poshyvanyk, "Rclinker: Automated linking of issue reports and commits leveraging rich contextual information," in *2015 IEEE 23rd International Conference on Program Comprehension*. IEEE, 2015, pp. 36–47.

[6] Y. Sun, Q. Wang, and Y. Yang, "Frlink: Improving the recovery of missing issue-commit links by revisiting file relevance," *Information and Software Technology*, vol. 84, pp. 33–47, 2017.

[7] S. L. Abebe, N. Ali, and A. E. Hassan, "An empirical study of software release notes," *Empirical Software Engineering*, vol. 21, no. 3, pp. 1107–1142, 2016.

[8] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, A. Marcus, and G. Canfora, "Arena: an approach for the automated generation of release notes," *IEEE Transactions on Software Engineering*, vol. 43, no. 2, pp. 106–127, 2016.

[9] ——, "Automatic generation of release notes," in *Proceedings of the 22nd acm sigsoft international symposium on foundations of software engineering*, 2014, pp. 484–495.

[10] M. V. Mäntylä, B. Adams, F. Khomh, E. Engström, and K. Petersen, "On rapid releases and software testing: a case study and a semi-systematic literature review," *Empirical Software Engineering*, vol. 20, no. 5, pp. 1384–1425, 2015.

[11] C. S and S. B, "Pro git," 2014.

[12] GitHub, "Comparing releases - github docs," 2020. [Online]. Available: https://docs.github.com/en/freepro-team@latest/github/administering-arepository/comparing-releases

[13] D. Wijayasekara, M. Manic, J. L. Wright, and M. McQueen, "Mining bug databases for unidentified software vulnerabilities," in *2012 5th International conference on human system interactions*. IEEE, 2012, pp. 89–96.

[14] C. Xiao, A. Sarabi, Y. Liu, B. Li, M. Liu, and T. Dumitras, "From patching delays to infection symptoms: Using risk profiles for an early discovery of vulnerabilities exploited in the wild," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 903–918.

[15] X. Feng, X. Liao, X. Wang, H. Wang, Q. Li, K. Yang, H. Zhu, and L. Sun, "Understanding and securing device vulnerabilities through automated bug report analysis," in *SEC'19: Proceedings of the 28th USENIX Conference on Security Symposium*, 2019, p. 887–903.

[16] F. C. do Rego Pinto, B. Costa, and L. Murta, "Assessing time-based and range-based strategies for commit assignment to releases," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2021, pp. 142–153.

[17] J. Kim, S. Kang, and J. Lee, "A comparison of software product line traceability approaches from end-to-end traceability perspectives," *International Journal of Software Engineering and Knowledge Engineering*, vol. 24, no. 04, pp. 677–714, 2014.

[18] M. F. Bashir and M. A. Qadir, "Traceability techniques: A critical study," in *2006 IEEE International Multitopic Conference*. IEEE, 2006, pp. 265–268.

[19] I. Galvao and A. Goknil, "Survey of traceability approaches in model-driven engineering," in *11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007)*. IEEE, 2007, pp. 313–313.

[20] K. Jaber, B. Sharif, and C. Liu, "A study on the effect of traceability links in software maintenance," *IEEE Access*, vol. 1, pp. 726–741, 2013.

[21] D. Meedeniya, I. Rubasinghe, and I. Perera, "Traceability establishment and visualization of software artefacts in devops practice: a survey," *International Journal of Advanced Computer Science and Applications*, vol. 10, no. 7, pp. 66–76, 2019.

[22] E. Bouillon, P. Mäder, and I. Philippow, "A survey on usage scenarios for requirements traceability in practice," in *International Working Conference on Requirements Engineering: Foundation for Software Quality*. Springer, 2013, pp. 158–173.

[23] F. Khomh, T. Dhaliwal, Y. Zou, and B. Adams, "Do faster releases improve software quality? an empirical case study of mozilla firefox," in *2012 9th IEEE working conference on mining software repositories (MSR)*. IEEE, 2012, pp. 179–188.

[24] M. V. Mäntylä, F. Khomh, B. Adams, E. Engström, and K. Petersen, "On rapid releases and software testing," in *2013 IEEE International Conference on Software Maintenance*. IEEE, 2013, pp. 20–29.

[25] S. Clark, M. Collis, M. Blaze, and J. M. Smith, "Moving targets: Security and rapid-release in firefox," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 1256–1266.

[26] R. Souza, C. Chavez, and R. A. Bittencourt, "Do rapid releases affect bug reopening? a case study of firefox," in *2014 Brazilian Symposium on Software Engineering*. IEEE, 2014, pp. 31–40.

[27] ——, "Rapid releases and patch backouts: A software analytics approach," *IEEE Software*, vol. 32, no. 2, pp. 89–96, 2015.

[28] T. Dhaliwal, F. Khomh, Y. Zou, and A. E. Hassan, "Recovering commit dependencies for selective code integration in software product lines," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2012, pp. 202–211.

[29] H. Kagdi, J. I. Maletic, and B. Sharif, "Mining software repositories for traceability links," in *15th IEEE International Conference on Program Comprehension (ICPC'07)*. IEEE, 2007, pp. 145–154.