Same file, different changes: The potential of meta-maintenance on GitHub

Hideaki HATA

Raula KULA

Takashi ISHIO

Christoph TREUDE
Singapore Management University, ctreude@smu.edu.sg

Same File, Different Changes: The Potential of Meta-Maintenance on GitHub

Hideaki Hata,* Raula Gaikovina Kula,* Takashi Ishio,* Christoph Treude†
*Nara Institute of Science and Technology
{hata, raula-k, ishio}@is.naist.jp
†University of Adelaide
christoph.treude@adelaide.edu.au

Abstract—**Online collaboration platforms such as GitHub have provided software developers with the ability to easily reuse and share code between repositories. With clone-and-own and forking becoming prevalent, maintaining these shared files is important, especially for keeping the most up-to-date version of reused code. Different to related work, we propose the concept of meta-maintenance—i.e., tracking how the same files evolve in different repositories with the aim to provide useful maintenance opportunities to those files. We conduct an exploratory study by analyzing repositories from seven different programming languages to explore the potential of meta-maintenance. Our results indicate that a majority of active repositories on GitHub contains at least one file which is also present in another repository, and that a significant minority of these files are maintained differently in the different repositories which contain them. We manually analyzed a representative sample of shared files and their variants to understand which changes might be useful for meta-maintenance. Our findings support the potential of meta-maintenance and open up avenues for future work to capitalize on this potential.**

I. INTRODUCTION

Clone-and-own is a quick way to create customized variants of a software project by copying an existing product and adapting it to a new set of requirements [1]–[3]. Despite perceived benefits, such as simplicity, availability, and independence of developers [2], clone-and-own has been criticized for leading to a large number of code clones [4], making it difficult to propagate changes such as bug fixes from one instance to another. Clone-and-own also leads to problems related to awareness: developers do not know when and where their code is being cloned, they do not know the origin of the cloned code, and they have no means of staying aware of changes to other instances that might benefit their own instance of the cloned code.

Source code reuse across multiple software projects has been widely studied in code clone research. It is reported that a large number of projects included copies of libraries [5]–[7]. Gharehyazie et al. reported that most projects obtain files from outside rather than providing their files to other projects [8]. Forking, including traditional hard forking [9] and recent fork-based development in GitHub [10], [11], is one type of source code reuse. Recent studies discussed several problems in forking, such as redundant development, lost contributions, and fragmented communities [12], [13].

Even with its large-scale code resources and a large amount of developers, Google is reported to address reuse properly with a monolithic source code management system [14], [15]. Such a monolithic system has several advantages: unified versioning, extensive code sharing and reuse, simplified dependency management, large-scale refactoring, and so on [14]. Based on the monolithic source code management, changes to core libraries are promptly and easily propagated through the dependency chain into the final products that rely on the libraries [14].

Although introducing such a complete monolithic source code management system in the entire free/libre open source software (FLOSS) ecosystem is not practical, tracking source code across multiple projects could bring new insight into software maintenance. If we can aggregate and compose useful changes from various repositories maintaining the same origins, sharing such composed changes can help software maintenance activities efficiently for multiple projects. We call this approach *meta-maintenance*.

To investigate the feasibility of meta-maintenance, in this paper, we report on the results of an exploratory study of almost 28 million files that are shared by multiple GitHub repositories. We found that in more than 70% of the repositories in our sample, there is at least one file which also exists in another repository. Most of these files have not been maintained, but there is a significant minority which has not only been maintained but often has received project-specific changes, such as bug fixes. We manually analyzed a representative sample of such files as part of a qualitative analysis, and found that files that are shared by a large number of repositories are often libraries (e.g., jQuery) while files that are shared by a smaller group of repositories tend to contain utility functionality (e.g., drivers). In the former case, the repositories which share the file tend to be unrelated while in the latter case, there is often a relationship between repositories (e.g., one repository relying on the code in another).

Investigating how files changed in different repositories revealed a number of different types of changes, including library updates (e.g., upgrading the jQuery library), commits taken from a known origin (e.g., the Linux kernel project), as well as project-specific changes (e.g., bug fixes). We argue that project-specific changes have the largest potential for meta-maintenance, and we conducted a survey in which

we asked project maintainers what they thought of specific instances where meta-maintenance could be applied in their repositories, i.e., the maintainers were maintaining a file that had been changed in another repository, potentially incorporating interesting changes. This survey result provided further evidence for the potential of meta-maintenance and pointed out interesting areas for future work.

II. RELATED WORK

Before outlining our methodology, we discuss extensive literature related to code clones, origin analysis, and forks.

A. Cross-Project Code Clones

Code clone detection is the most popular approach to analyze source code reuse activities. Since developers may modify a code fragment for their own purpose, various tools have been proposed to detect similar source code fragments [16]. Kamiya et al. [17] proposed CCFinder that analyzes normalized token sequences. Jiang et al. [18] proposed DECKARD that compares a vector representation of an abstract syntax tree. Nguyen et al. [19] proposed Exas that compares a vector representation of a dependence graph. Sasaki et al. [20] proposed FCFinder that recognizes file-level clones using hash values of normalized source files. Cordy et al. [21] proposed NiCad that compares a pair of code blocks using a longest common subsequence algorithm. Sajnani et al. [22] proposed SourcererCC that compares a pair of code blocks using Jaccard index of tokens.

Code clone detection tools revealed that software developers often copy source files from other projects. Hemel et al. [23] analyzed vendor-specific versions of Linux kernel. Their analysis showed that each vendor created a variant of Linux kernel and customized many files in the variant. Ossher et al. [5] analyzed cloned files across repositories using a lightweight technique. They reported that projects cloned files from related projects, libraries, and utilities. Koschke et al. [24] also reported that a relatively large number of projects included copies of libraries. Lopes et al. [7] analyzed duplicated files in 4.5 million projects hosted on GitHub and reported that the projects have a large amount of file copies. Gharehyazie et al. [8] analyzed cross-project code clones of 5,753 Java projects on GitHub. They also analyzed timestamps of the clones and reported that developers often copy an entire library, and some projects serve as hubs (sources) of clones to other projects. The analyzed source code is a snapshot at a certain point of time. Our study extends the analysis by including additional programming languages and all the versions of projects.

Identified code reuses across projects can be used for several applications. Dang et al. [25] reported that Microsoft developers use code clone information to fix bugs in multiple products at once. Rubin et al. [26] reported that industrial developers extract reusable components as core assets from existing software products. Bauer et al. [27] proposed to extract code clones across products as a candidate of a new library. Ishihara et al. [28] proposed a function-level clone detection to identify reusable functions in a number of projects. Luo et al. [29] proposed a method to identify semantically equivalent basic blocks for code plagiarism detection. Chen et al. [30] proposed a technique to detect clones of Android applications using similarity between control-flow graphs of methods.

Davies et al. [31], [32] proposed a file signature to identify the origin of a jar file using classes and their methods in the file ignoring the details of code. Ishio et al. [3] extended the analysis to automatically identify libraries copied in a product. Similar to these approaches, this study starts the analysis from file-level clones.

B. Origin Analysis

Software projects use source code repositories to manage the versions of source code. Although a repository tracks modified lines of code between two consecutive versions of a file, the feature is not always sufficient to represent a complicated change. Godfrey et al. [33] proposed origin analysis to identify merged and split functions between two versions of source code. The method compares identifiers used in functions to identify original functions. Steidl et al. [34] proposed to detect source code move, copy, and merge in a source code repository. The method identifies a similar file in a repository as a candidate of an original version. Kawamitsu et al. [35] proposed an extension of origin analysis across two source code repositories. Their method identifies an original version of source code in a library's source code repository. Spinellis [36] constructed a Git repository including the entire history of Unix versions. The unified repository enables developers to investigate the change history of source files across projects. German et al. [37] used CCFinder to detect code siblings reused across FreeBSD, OpenBSD and Linux kernels, and then investigated the source code repositories of the projects to identify the original project of a code sibling. Krinke et al. [38] proposed to distinguish copies from originals by comparing timestamps of code fragments recorded in source code repositories. Krinke et al. [39] used the approach and visualized source code reuse among GNOME Desktop Suite projects. In this study, we do not intend to identify origins but investigate files in clone sets in terms of whether there are useful changes in their histories that could benefit other repositories.

C. Forks

Software developers often fork repositories in order to propose source code changes to the original projects. Stanciulescu et al. [40] analyzed an OSS community and reported that forks contribute new features while developers may spend their effort on redundant development. Ren et al. [41] proposed a machine learning model using change description, patch content, and issue tracker to identify redundant code changes in forks. Zhou et al. [12] proposed a tool named Infox to automatically identify unique source code changes as new features in forks. Different from those analyses, this study does not focus on forked repositories, and revealed that non-forked repositories also include their own changes to reused source

code. Zhou et al. [13] reported that better modularity and centralized management are associated with more contributions and a higher fraction of accepted pull requests from forks. They also reported that the lower the pull request acceptance rate, the higher the chance of a project having hard forks. Regarding hard forking, Ray et al. analyzed porting of an existing feature or bug fix across forked projects [9]. They reported that forking allows independent evolution but results in the significant cost of porting activity.

Propagating changes to others is also considered as a challenge in software product line engineering. Strüber et al. [42] included the task in scenarios for evaluating techniques that support developers during the evolution of variant-rich systems. To support such propagation of a bug fix, ReDeBug [43] and Clorifi [44] have been proposed to efficiently identify source file clones to which a patch should be applied.

III. META-MAINTENANCE: TERMINOLOGY

The concept of meta-maintenance is based on the model of individual evolution of the same files (clones) in different repositories (not limited to forks), which is inspired by Google's monolithic source code management system [14], with the aim to maintain the overall ecosystem. We now define the terminology needed to describe meta-maintenance at file level.

A **seed file** is a specific version of a file that is shared in multiple software repositories. In this paper, we focus on Git repositories, thereby considering specific Git blobs appearing in multiple repositories to be seed files. A Git blob stores the content of a file with a specific version, and is identified with its SHA-1 hash of the content [45]. Let f refer to a seed file for a repository r so that $r(f)$ indicates the repository r containing file f. Software repositories that contain the same seed file are then referred to as belonging to the same **seed family** SF_0, where $SF_0 = \{r_1(f), r_2(f), ..., r_n(f)\}$ for repositories r_1, r_2 to r_n. After the time T passed, the seed family SF_0 becomes $SF_T = \{r_1(f_x), r_2(f_y), ..., r_n(f_z)\}$, where f_x, f_y, and f_z represent updated files from the same seed file f, and x, y, and z are the number of changes to the files. The file $f_0 = f$, and we call $f_i(i > 0)$ a **variant**. We call f_l and f_m **duplicate** variants if their contents are the same even if l and m are different. Meta-maintenance involves the analysis of how variants in the same seed family evolve.

Forking is an explicit form of sharing seed files. In this paper we do not focus on forks as seed families. Recent studies have tried supporting change propagation in forks with a centralized model, that is, accumulating changes upstream and distributing them downstream [12], [13], [41]. As well as such explicit reuse relationships, meta-maintenance is conceptualized to be able to support implicit reuse relationships. We intend to support repositories in seed families with a decentralized model, that is, useful changes are aggregated from individual repositories and are distributed to others.

IV. PREPARATIONS

In this section, we present our research questions and data collection methodology.

A. Research Questions

The main goal of the study is to explore the potential for meta-maintenance for contemporary software projects in GitHub. Based on this goal, we constructed six research questions to guide our study. We now present each of these questions, along with our motivation.

(RQ1): *How prevalent are seed files in software repositories?*
(RQ2): *What kinds of seed files are there?*
(RQ3): *Are there relationships among repositories in seed families?*
(RQ4): *What was the main driver of the changes for variants?*
(RQ5): *How uniquely do variants evolve in seed families?*
(RQ6): *How do developers consider changes for variants?*

The motivation of **RQ1** is to understand whether seed files and their seed families are common in the wild. Furthermore, we would like to quantitatively explore the distribution, maintenance statuses, and patterns of seed families. **RQ2**, **RQ3** and **RQ4** require a deeper analysis of seed files and repositories, where we would like to understand the nature of the seed files and reasons behind their evolution. The key motivation for **RQ2** is to identify kinds of seed files that are used in the software repositories. Such insights would help identify the nature for why these seed files were reused in other repositories. Then for **RQ3**, we would like to understand whether there is a connection among repositories in the same seed families. The key motivation for **RQ4** is to determine the key drivers that influenced the changes that were made to the seed files in those repositories. **RQ5** then investigates the seed files from an evolutionary and maintenance standpoint. We would like to quantitatively understand how developers are updating or maintaining variants after reusing seed files in their projects and how differently variants evolve. Our aim for **RQ6** is to understand how developers react to changes in other variants.

B. Data Collection

Here we describe our procedures for target repository preparation, Git blob extraction, and stratified sampling, for our data collection.

a) Repository preparation: To pursue the feasibility of meta-maintenance, we collect a large amount of software development repositories that have been actively developed. We follow the same procedure as in a previous study [46] to identify candidate repositories. We target software development repositories on GitHub written in seven *common* programming languages, that is, C, C++, Java, JavaScript, Python, PHP, and Ruby. These languages have been ranked consistently in the top 10 languages on GitHub from 2008 to 2018 (based on the number of repositories from 2008 to 2015 [47], the number of pull requests from 2014 to 2018 [48], and top languages from 2014 to 2018 in the official report [49]). Using

TABLE I: Collected repositories

language	# candidates	# obtained (%)
C	3,262	2,962 (91%)
C++	4,219	3,824 (91%)
Java	5,911	5,427 (92%)
JavaScript	9,017	7,960 (88%)
Python	6,606	6,087 (92%)
PHP	3,877	3,388 (87%)
Ruby	2,639	2,359 (89%)
sum	**35,531**	**32,007 (90%)**

Fig. 1: Plot of the number of seed families for each number of seed files (log-log scale).

TABLE II: Construction of the stratified sample

	# seed families	sample size
common	662	243
sometimes	2,132,141	384
rare	25,861,784	384
sum	**27,994,587**	**1,011**

the GHTorrent dataset[1] [50], we identify active repositories for the seven languages with the following criteria [46]: (i) having more than 500 commits in their entire history (the same threshold used in previous work [51]), and (ii) having at least 100 commits in the most active two years to remove long-term less active repositories and short-term projects that have not been maintained for long. We determine repositories' languages based on the GHTorrent information. As mentioned in Section III, we exclude forked repositories in this study. We remove repositories that had been recorded in GHTorrent as forks of other repositories. Note that even though we exclude such explicit forks, which have been targeted extensively by related work [13], there can be implicit forks in our dataset. This could be because repositories were forked outside of GitHub, for example. Although these implicit forks are not considered in research on fork-based development [12], [13], [41], we keep these repositories in this study to investigate the feasibility of meta-maintenance. With the above procedure, we obtained the candidate list of repositories for the seven languages as shown in Table I. From the candidate list, some repositories were not available because they had been deleted or made private. Additionally we exclude repositories that have only one committer in order to remove self-learning repositories (online judge code, for example). In total, we obtained more than 32,000 repositories, which is almost 90% of the candidate repositories.

b) Git blob extraction: From each repository, we extracted all existing blobs and their file names using `git rev-list -all -objects` and `git cat-file` commands. Only the blobs of source files with the following file extensions are targeted in this study: `.c`, `.h` (C), `.cc`, `.cp`, `.cpp`, `.cx`, `.cxx`, `.c++`, `.hh`, `.hp`, `.hpp`, `.hxx`, `.h++` (C++), `.java` (Java), `.js` (JavaScript), `.py` (Python), `.php` (PHP), and `.rb` (Ruby). Blobs appearing in multiple repositories are considered to be seed files. In total, we obtained 27,994,587 seed files in our dataset.

c) Stratified sampling: To understand these seed files and their characteristics towards answering our research questions, we conducted stratified sampling to enable us to gain insights into a variety of different scenarios. We hypothesize that the size of the corresponding seed families is a particularly important criterion for distinguishing different kinds of seed
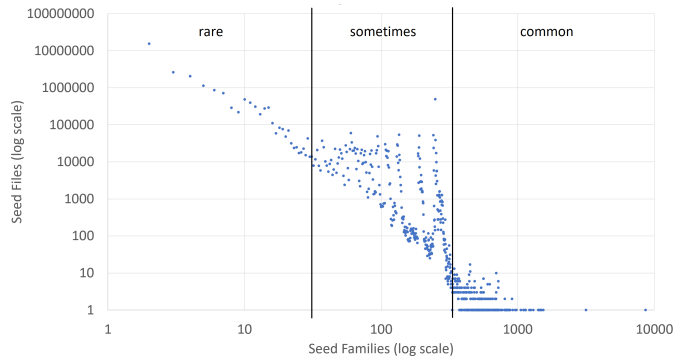
files. To understand the distribution of size of seed families across seed files, we plotted the distribution, cf. Figure 1. Each dot represents a tuple of <*number of seed families, size of seed family*>, e.g., the left-most dot indicates that there were more than 10 million seed families of size two. We then divided seed files into three categories: those that are part of big seed families ("common"), those are part of small seed families ("rare"), and an additional category in between ("sometimes"). Based on visual inspection of the plot in Figure 1, we set the thresholds at a family size of at least 331 for common, and at least 28 for sometimes. The left side of Figure 1 represents small seed families (rare) with less than 28 instances each and the right side represents large seed families (common) with more than 330 instances each. Table II shows the corresponding numbers. The stratum with a family size of at least 331 contains 662 seed families, the stratum with a family size of at least 28 (but no more than 330) contains more than 2 million seed families, and the stratum with a family size smaller than 28 contains more than 25 million seed families. The goal of our stratified sampling is to understand these different groups better. We randomly sampled a statistically representative number of seed families from each stratum, ensuring that our conclusions regarding ratios within each sample would generalize to the entire stratum with a confidence level of 95% and a confidence interval of 5. The last column of Table II shows the number of seed families in each sample, for a total sample size of 1,011.

For the 1,011 seed families, the evolutionary period of variants within a seed family, measured from the earliest commit in seed files to the latest commit in variants, was a minimum of 0.4 years, a median of 5.5 years, and a maximum of 18 years.

---

[1]MySQL database dump 2019-02-01 from http://ghtorrent.org/downloads.html.

V. Method

We describe our mixed-method procedure including a quantitative analysis, a qualitative analysis, and a survey.

A. Quantitative Analysis

To understand the prevalence of seed files in a large amount of software development repositories (**RQ1**), we conduct a quantitative analysis of our collected dataset and stratified sample in terms of existence, status, size, and modifications of seed files. The history of a variant is examined based on the path of the seed file. Therefore, variants that have changed significantly in content can be tracked, but variants that have been renamed or moved are not tracked in this study.

To investigate how variants evolve in different projects (**RQ5**), we measure the degree of variant evolution in each family using two metrics: $Retention$ of similarities with seed files and $Uniqueness$ of differences with other variants.

Retention: how variants are similar to seeds. For a pair of a variant v and its seed file s, we measure their similarity as follows, similar to a previous study of clone-and-own [3].

$$Retention_f(v, s) = \frac{|\{\tau(v) \cap \tau(s)\}|}{|\tau(v)|}$$

where $\tau(x)$ represents a set of trigrams of tokens in file x ignoring comments and white spaces. To recognize tokens and comments in source code, we employed lexical analyzers based on the Ripper library for Ruby and ANTLR4 for the other six languages. We calculate an average retention value for all variants V in a seed family as follows.

$$Retention(V, s) = \frac{\sum_{v \in V} Retention_f(v, s)}{|V|}$$

A higher $Retention(V, s)$ means variants are similar to their seed files, that is, variants have not been changed largely.

Uniqueness: how variants evolved differently. To measure the amount of project-specific changes for each variant, we calculate the uniqueness of content as follows.

$$Uniqueness(V, s) = \frac{\sum_{v \in V} u(v, \{s\} \cup V \setminus \{v\})}{|V|}$$

$$u(v, F) = \frac{|\{t \in \tau(v) \mid \forall f \in F. t \notin \tau(f)\}|}{|\tau(v)|}$$

The function $u(v, F)$ measures the amount of trigrams only in a variant v in the family. A higher $Uniqueness(V, s)$ value indicates that the variants are more different from one another.

B. Qualitative analysis

To understand the types of seed files (**RQ2**), the relationships among repositories in seed families (**RQ3**), and the characteristics of changes for seed files (**RQ4**), we manually annotate seed files and seed families in our sample, which is conducted in multiple iterations. In each iteration except for the last one, three authors independently annotate instances from each stratum using open coding. After each iteration, the authors discuss the codes that have emerged and how to distinguish between them. We repeat this process on new subsets from each stratum until we finalize the list of codes and achieve Kappa agreement of at least 0.7 among the three annotators. One author then annotates the rest of the data. Since there can be a large number of variants in a seed family, we annotate up to five variants per seed family. If a seed family contains more than five variants, we conduct stratified sampling by selecting the three variants with the largest number of commits and the two variants with the lowest number of commits.

C. Survey

To understand the potential of meta-maintenance (**RQ6**), we conduct a survey for developer feedback. To find meta-maintenance opportunities, we searched unique commits in seed families, that is, we identified commits that appear only in single repositories. To limit the search space, we target commits whose commit message contains the keyword '*fix*', which is considered to be related to fixing bugs. There are 26, 81, and 135 seed families that contain at least one unique commit for rare, sometimes, and common samples, respectively. Again, to limit the search space, seed families with more than two unique commits are filtered out, which resulted in 11, 51, and 69 seed families for rare, sometimes, and common samples, respectively. We manually investigate these seed families. By checking the latest commits on GitHub, we found that some repositories applied the identified unique commits later by cherry-picking. We also found that some unique commits are not easily applicable to other repositories because of the large differences in the histories.

Five cases were selected where the changes were neither too large nor too complex. The survey was distributed in forums, issues, or emails of the corresponding projects. As part of the questionnaire, we asked (a) how important the seed file was to the project, (b) what kind of maintenance activities were the developers interested in regarding the file, and (c) whether they would be interested in a meta-maintenance approach. We received three responses and noticed that due to specificity of the files, only the core contributors of the projects responded in all cases.

VI. Findings

Here we present our findings for each research question.

A. Prevalence of Seed Files (RQ1)

a) Seed file existence: Figure 2a shows the percentages of repositories that have at least one seed file. Although the percentage is relatively low for Java (but still more than 70%), more than 80% of repositories contain seed files for the other languages. Especially for repositories written in C, JavaScript, PHP, and Python, more than 90% of repositories contain seed files. Figure 2b presents the distributions of the number of different seed files per repository, for repositories that have at least one seed file. Median values are shown in the boxplot. We
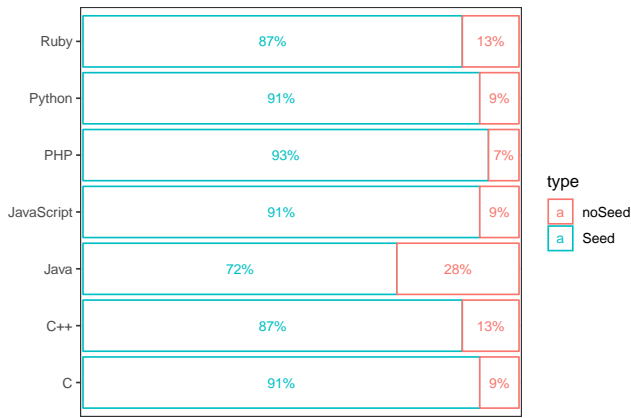
(a) Percentage of repositories with and without seed files.
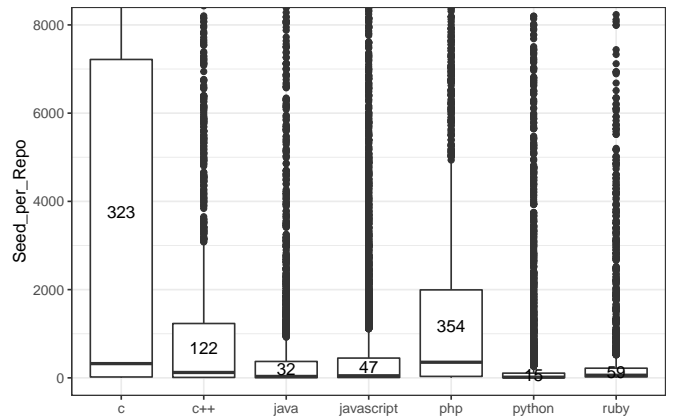
(b) Distribution of the number of seed files per repository.

Fig. 2: Prevalence of seeds from (a) seed file existence and (b) the number of seed files in a repository, per language.

TABLE III: Frequency of variant statuses in our sample

	common		sometimes		rare	
dormant	22,937	(18%)	13,290	(26%)	182	(13%)
inactive	83,517	(65%)	17,266	(33%)	671	(47%)
unchanged	15,435	(12%)	1,278	(2%)	52	(4%)
maintained	7,530	(6%)	20,181	(39%)	518	(36%)
sum	**129,419(100%)**		**52,015(100%)**		**1,423(100%)**	

TABLE IV: Frequency of seed family types in our sample

	common		sometimes		rare	
not maintained	0	(0%)	48	(13%)	240	(63%)
empty seed	10	(4%)	0	(0%)	0	(0%)
zero variance	132	(54%)	163	(42%)	128	(33%)
non-zero variance	101	(42%)	173	(45%)	16	(4%)
sum	**243(100%)**		**384(100%)**		**384(100%)**	

see that repositories contain dozens of seed files (at median), which implies there likely exist sets of seed files commonly shared by multiple repositories. For C and PHP, whose projects have more seed files, we see forked projects from specific products, which should share many seed files.

b) Status of variants: For the concept of meta-maintenance, we are interested in only repositories that are maintaining variants. We distinguish variants based on their statuses as follows.

dormant: a variant is in a dormant or an unmaintained repository. Similar to previous studies [52], [53], we set one year as a threshold to consider dormant, that is, we consider a repository dormant or unmaintained if the repository does not have commits in 2018.

inactive: a variant does not exist in the main branch (usually *master* in GitHub).

unchanged: the seed file has not been modified and exists in the latest commit of the main branch.

maintained: a variant had been changed and exists in the latest commit of the main branch.

Table III shows the frequencies of the variant statuses in the three strata of our sample. We see that the majority of variants are not *maintained*, 94% for common, 61% for sometimes, and 64% for rare.

c) Types of seed families: Table IV summarizes the frequencies of seed family types as described below.

not maintained: We observed that some projects recorded in GHTorrent point to the same repositories even though ID

and/or project names are different. This happens when repositories re-registered in GitHub. We consider a seed family not maintained if there is no maintained variant in the seed family after removing identical repositories, that is, all variants in the seed family are either dormant, inactive, unchanged, or identical.

empty seed: We found some seed files do not have meaningful contents as source code, such as no line, only a blank line, or only comment lines. We manually identified those files. As seen in Table IV, such empty seed files appear only in the common stratum.

zero variance: There is only one *maintained* variant or the same changes have been applied to all variants, that is, *duplicate* variants. We found some variants had been cherry-picking commits from their origins with different frequencies.

non-zero variance: Even after removing duplicate variants, there are multiple *maintained* variants in a seed family.

For meta-maintenance, we are only interested in variants that have evolved independently. Therefore, only seed families of *non-zero variance* are targeted in our study. In the seed families, duplicate variants are removed for the further analyses. Table V shows the number of seed families and the number of remaining variants in our filtered sample. This sample is used to answer the following research questions.

TABLE V: Target data in our sample

	# seed families	# variants
common	101	897
sometimes	173	1,748
rare	16	47
sum	**290**	**2,692**

TABLE VI: Types of Seed Files

	library	configuration	utility	other
common	91	5	2	3
sometimes	5	1	157	10
rare	1	1	12	2

Fig. 3: LOC in seed files, per stratum.

Summary: We revealed that seed files are prevalent. In more than 70% of the targeted 32,007 repositories in GitHub, there exists at least one seed file. Despite the large amount of seed files, most of them have not been maintained nor used in the latest snapshots. However, there exists some amount of potential variants for meta-maintenance for each stratum.

B. Types of Seed Files (RQ2)

We achieved Kappa agreement of 0.7 in our annotation. Our analysis revealed the following four types of files, with Table VI showing the number of instances for each code in each stratum:

library: Seed files which contain a library (a program that contains a collection of code used by applications) are particularly widespread in the common stratum where they account for 91% of all seed files that we encountered during our annotation, indicating that most seed families in the common stratum are library users. A representative example is the jQuery library, e.g., in the modxcms-jp/evolution-jp repository.[2]

utility functionality: Seed files which contain utility functionality (a system software for controlling the operation of a computer, devices, etc.) were predominantly found in the sometimes stratum where they account for 91% of the annotated seed files—coincidentally the same ratio as library files in the common stratum. Drivers, such as the driver for Avance Logic ALS300/ALS300+ soundcards in the masahir0y/linux repository[3] are a representative example for this group of seed files.

configuration: Configuration files are much less common as seed files, with most of them appearing in the common strata. The config.php file of the contao/core-bundle[4] repository can serve as an example.

other: We used the code "other" mostly for header files or files containing version information, such as version.php in the gMagicScott/core.wordpress-mirror repository.[5]

We also investigated typical file sizes in each stratum. Figure 3 shows the corresponding distributions. In general, the number of lines of code for libraries, which are widespread in the common stratum, is smaller compared to the other strata, mostly due to presence of a large number of minified JavaScript libraries. The differences between sometimes and rare strata are negligible.

Summary: We revealed that seed files which are part of large seed families are often libraries, whereas seed files from medium-sized seed families tend to contain utility functionality. Files from large seed families tend to be smaller in terms of number of lines of code than files from smaller seed families.

C. Repositories in Seed Families (RQ3)

As documented by previous work [54], GitHub hosts a wide variety of projects, the characterization of which is beyond the scope of this paper. Instead, to understand the potential of meta-maintenance, our focus is on the relationships among repositories in seed families, i.e., whether there is a connection among them. Such connections could stem from forking or copying as well as from the code of one repository using the code from another repository. We distinguish each repository in our sample into "related" or "non-related". In this annotation, we achieved perfect Kappa agreement (i.e., 1.0) among the three annotators.

related: There is an explicit relationship among repositories, e.g., one is a fork of another, their names are similar or identical, or because one mentions the other prominently in its documentation. The most common example of such a relationship are the many Linux variants in our sample. For example, the driver file gl520sm.c is contained in the repositories masahir0y/linux[6] and Whissi/linux-stable.[7] The repositories can easily be connected based on their names, even though only the former contains the information that it is forked from torvalds/linux. In

---

[2]https://github.com/modxcms-jp/evolution-jp/blob/0dce7db4/manager/media/script/jquery/jquery.min.js

[3]https://github.com/masahir0y/linux/blob/dc4060a5dc25/sound/pci/als300.c

[4]https://github.com/contao/core-bundle/blob/1373ebc29/src/Resources/contao/config/config.php

[5]https://github.com/gMagicScott/core.wordpress-mirror/blob/8548899126/wp-includes/version.php

[6]https://github.com/masahir0y/linux

[7]https://github.com/Whissi/linux-stable

TABLE VII: Relationships among Seed Families

	related	non-related
common	0	398
sometimes	624	24
rare	40	5

TABLE VIII: Change Types

	known origin	library updates	project-specific	tangled updates	other
common	9	202	96	15	76
sometimes	619	11	11	3	4
rare	34	0	8	0	3

addition, the latter repository states in its description that it is a mirror of the Linux distribution hosted on git.kernel.org.[8]

non-related: On the other hand, many seed families do not contain any evidence to suggest that there is a connection among the repositories, apart from using at least one common file. For example, the repositories MoveLab/tigatrapp-server[9] and magda-io/magda[10] both contain Twitter's Bootstrap library via a bootstrap.js file. The former repository contains a server with which Tigatrapp (a Spanish citizen science project) apps communicate whereas the latter contains an open-source software platform designed to assist in all areas of the data ecosystem. Apart from both repositories using Bootstrap, there is no apparent connection among the repositories. Note that both repositories have made one change to their instance of bootstrap.js since the files were identical.

Table VII shows that the ratio of related vs. non-related differs significantly between the strata. While all repositories in our sample from the common stratum were not related, the vast majority from sometimes and rare were related.

> **Summary**: Repositories which contain seed files that are part of large seed families tend to not be related. In many cases, they are different repositories using the same library. On the other hand, repositories which contain seed files which are part of smaller seed families tend to be related.

D. Changes for Seed Files (RQ4)

To understand how the repositories might be able to benefit from meta-maintenance, we investigated the commit histories of all files in our sample to characterize the set of changes that had been applied to them. Our annotation revealed four categories plus other, and we achieved Kappa agreement of 0.76.

reference to a known origin: For many repositories in our sample, the origin is obvious—this applies in particular

to the various Linux variants. In those cases, changes that have been applied to seed files are often the same commits that have been applied to the origin. We were often able to detect this directly from the commit meta information, e.g., when commit author and commit committer are different. For example, the process.c file[11] in the Ziyann/omap repository attracted 104 commits since being identical across its seed family, many of which were committed by Linus Torvalds[12] and authored by a contributor of the Ziyann/omap repository.

library updates: Library updates are the most common kind of change in the common stratum, see Table VIII. An example is the previously mentioned jQuery library, e.g., in the modxcms-jp/evolution-jp[13] repository. While there have been 18 commits to this file since it was identical across its seed family, the commit messages of these commits[14] clearly show the pattern of library updates, e.g., "Update jQuery 1.11.0 –> 1.11.1" and "Update – jQuery 3.2.1".

project-specific changes: The most interesting case for meta-maintenance are project-specific changes which are not library updates or made in reference to a known origin. As Table VIII shows, we found such files in all strata. An example is the jQuery library[15] in the dmitrykuzmin/chat repository which has one new commit since being identical across its seed family with the commit message "Trying to fix IE issues.". We argue that such fix attempts might be relevant to other repositories containing the same file. We found project-specific changes in at least one quarter of seed families in the common and rare strata.

tangled updates: In case the commit history contained changes that could not easily be localized to the file under investigation, we applied the code "tangled updates". In these cases, meta-maintenance on the basis of files is unlikely going to be successful since changes affected many files—often hundreds if not thousands. Tangled updates were not widespread in our sample.

other: For change histories which did not fit any of the previous categories, we applied the code "other". An example are the commits to bootstrap.js[16] in the sparc-request/sparc-request repository. All of the four commits made since the file was identical across seed families indicate updates to the copyright information, most of them simply changing the year. This is unlikely to be useful for maintenance.
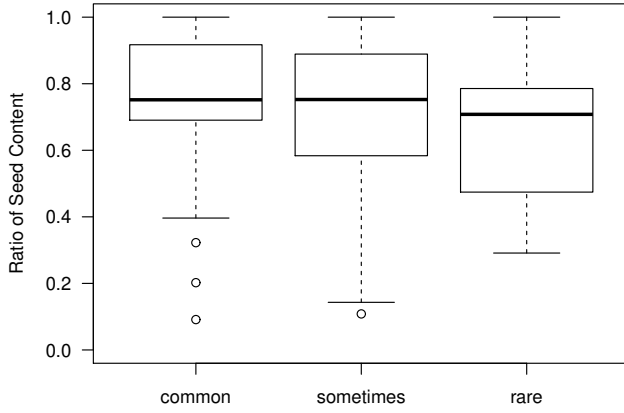
---

[8]https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git
[9]https://github.com/MoveLab/tigatrapp-server
[10]https://github.com/magda-io/magda

[11]https://github.com/Ziyann/omap/blob/7c07453808b/arch/powerpc/kernel/process.c
[12]https://github.com/torvalds
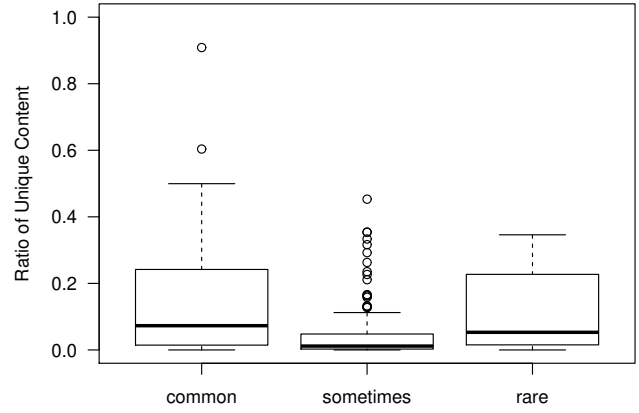[13]https://github.com/modxcms-jp/evolution-jp
[14]https://github.com/modxcms-jp/evolution-jp/commits/0dce7db475116f3a35206714e2721bf355f049c2/manager/media/script/jquery/jquery.min.js
[15]https://github.com/dmitrykuzmin/chat/blob/fa33c7e/webapp/src/main/webapp/lib/jquery-2.1.4.min.js
[16]https://github.com/sparc-request/sparc-request/blob/0eb424aba/app/assets/javascripts/bootstrap.js

(a) Distribution of $Retention(V, s)$, per stratum.

(b) Distribution of $Uniqueness(V, s)$, per stratum.

Fig. 4: Evolution of variants from (a) similarities with seed files and (b) differences with other variants.

Summary: While some variants were updated in references to a known origin or as part of regular library updates, in particular seed files that are part of large (common stratum) or small (rare stratum) seed families also contain a significant number of project-specific changes which could be useful for meta-maintenance.

Summary: Variants in the common and rare families contain project-specific changes in terms of unique tokens, while variants in the sometimes families tend to share common tokens. Although our analysis provides the basis for understanding how uniquely variants evolve, further research is needed to understand seed files in terms of evolution and maintenance.

E. Uniqueness of Variants (RQ5)

Figure 4a shows the distributions of the $Retention$ values for each stratum. Their medians are 0.75, 0.75, and 0.71, for common, sometimes, and rare, respectively. Most variants include some changes from the seed file. While the rare families seem to include more changes, the differences among the strata are not statistically significant. The Wilcoxon Rank Sum Test results in $p = 0.267$ for the rare-sometimes pair, $p = 0.065$ for the sometimes-common pair, and $p = 0.097$ for the rare-common pair.

Figure 4b shows the distributions of the $Uniqueness$ values for each stratum. The medians are 0.09, 0.01, and 0.07, for common, sometimes, and rare, respectively. The Wilcoxon Rank Sum Test shows that sometimes families have significantly lower uniqueness than rare and common families ($p = 0.005$ for the rare-sometimes pair and $p < 0.001$ for the sometimes-common pair). The difference between rare and common families is not significant ($p = 0.674$). The variants in the common and rare families include more unique changes. Those unique changes are potentially useful for other projects in the same family. For the sometimes families, variants tend to evolve by following the changes in their seed files but do not include much unique content. This result confirms the findings from our manual investigation (Section VI-D).

F. Developer Feedback on Meta-Maintenance (RQ6)

We present three cases where we received responses from developers.

Case Study 1: variants of `jQuery.js` library. jQuery is a widely used JavaScript library. Although the latest version in March 2020 is 3.4.1, some projects had been using older 1.x versions because of project-specific reasons. Joomla[17] is one such project which had applied bug fixing related to security issues.[18]

The survey was completed by an experienced core developer of Joomla, who in the comments thanked us for bringing up this topic. When asked about specific modifications to the same file in another repository, the respondent revealed that they are "*Mainly interesting for security reasons*" and gave us the following detailed comment.

> *For 3rd party components, we use tools like npm or composer to update them to the latest version. Only if we have to support a particular version we maintain the files our self and only for security fixes.*

This answer reveals an interesting use case for the concept of meta-maintenance: the adoption of security-related changes that have already been applied elsewhere. As a previous study reported, vulnerability issues are not appropriately addressed in many software development repositories [55]. Aggregating

[17]https://github.com/joomla/joomla-cms
[18]https://github.com/joomla/joomla-cms/commits/
a81ada410a5bf6b700a79d432fc5926146ac9f94/media/jui/js/jquery.js

appropriate fixes and distributing to individual repositories may support such cases.

Case Study 2: variants of `abc_object.cc` in Blender forks. Blender is an open source project for 3D computer graphics. In the rare families of our sample, there are two forks: Bforartists[19] and blender278.[20] The former is an active project merging all the recent commits from the origin without project-specific changes, while the latter does not have commits in the last half year but has project-specific changes from before. Both projects maintained the seed file `abc_object.cc` at least in 2018. After forking from the same content, the former had been changed by 31 commits and the latter had been changed by 13 commits.

The survey was completed by an experienced core developer of Bforartists, who claimed that the file is of high priority (5/5 on the Likert scale). When asked about modifications to the same file in another repository, the response was as follows.

> *Relevant is the Blender source code, since we regularly merge the newest changes from Blender source code. We are not interested in another fork with probably outdated code, or code that conflicts with our changes. ... maintaining code from others is always problematic. It's hard enough to keep our own changes working. We are a fork of Blender, not a fork of another fork..*

These comments point out important future work to make meta-maintenance a reality. To maintain seed files, some forked projects prefer to keep following changes in the original repository and do not consider applying changes from other repositories. This is a reasonable approach when centralized fork models work. We can easily identify repositories that are only following the original repository—meta-maintenance appears to be less relevant for them. In addition, we can consider aggregating project-specific changes and send them to the original repositories.

Case Study 3: variants of `zlib` library. zlib is a library for data compression, which is widely used in many projects. We observed a local change which fixes typos in one repository.[21] The survey was completed by the author of zlib. Although the developer was not interested in the suggested typo fix, he described the potential of applying commits from other repositories as:

> *They may have performance or other improvements.*

Summary: We learned from developers that supporting changes related to security and performance, among others, could be a promising use case for meta-maintenance and is desired by developers. Responses also point out future work to further understand the nature of relationships between repositories.

---

[19]https://github.com/Bforartists/Bforartists

[20]https://github.com/tangent-animation/blender278

[21]https://github.com/radareorg/radare2/commit/bc3425e73d294cbded877b66f0d60183edb5dd2e

VII. DISCUSSION

Based on our results, we now summarize the open challenges and barriers that need to be addressed before meta-maintenance can be fully realized. Then we discuss the limitations of our study.

A. Challenges to Meta-Maintenance

This paper establishes the state-of-the-practice for investigation of what type of meta maintenance is useful and under which conditions. To fully realize the potential of meta-maintenance, further research is required in the follow areas:

- *In-depth investigation of clone-and-own relationships in sets of seed files*. This study only focuses on single files as seeds. However, it is natural for software development projects to reuse a set of files. We need to develop techniques to identify seed families by considering a set of files, similar to a previous study [3]. During our manual investigation in **RQ6**, we observed that some seed families consist of sub-groups of repositories. Repositories in a group evolve similarly, and this results in large differences in the histories across different sub-groups. Measuring the similarities of histories (similar to the analysis in **RQ5**) is a promising and challenging area for future work.

- *Developing a global source code tracking system*, to understand relationships of repositories and the histories of seed files. This could be similar to Google's monolithic SCM system [14], [15], but in the ecosystem of open source software projects. The findings from **RQ1** reveal that many repositories could be connected within this global system.

- *Techniques to identify useful changes between variants*. There exist related techniques for fork-based development [12], [56] and software product lines [57]. Extending existing characterization studies, e.g., repeated bug fixes [58] to clone-and-own instances may provide further insights to develop such techniques. As learned from the findings in **RQ6**, changes related to security and performance are promising types to be identified as useful changes.

- *Tool support for meta-maintenance*. Tools to help developers find specific changes and maintain code automatically could be practically useful. Such tools could be based on push and pull models to aggregate and distribute useful changes to the ecosystem.

B. Threats to Validity

Threats to *construct validity* exist in our data collection procedure. Our criteria for selecting repositories may have ignored recent active projects and projects worked on by a single developer. Although we categorized repositories by programming language based on GHTorrent information, the information can be inaccurate. Further exploration to develop better criteria is needed. Threats to *external validity* exist in our repository preparation. Although we analyzed a large amount of repositories on GitHub, we cannot generalize our

findings to industrial projects nor FLOSS in general; some FLOSS repositories are hosted outside of GitHub, e.g., on GitLab or private servers. In addition, empirical studies are needed for other programming languages. To mitigate threats to *reliability*, we prepare an online appendix of our studied dataset with associated information (see Section IX).

VIII. CONCLUSION

To explore the potential of meta-maintenance, we conducted an exploratory study with (i) a quantitative analysis of 27,994,587 seed files from 32,007 Git repositories to establish the prevalence of seed files, the extent to which seeds evolve, and the uniqueness of seeds; (ii) a qualitative analysis of a stratified sample of 1,011 seed files to determine the kinds of seeds, the relationships among seed families, and main drivers for changes in the variants; and (iii) a survey for developer feedback.

Our work shows the potential of meta-maintenance with an extensive type of changes identified other than simple forking. Based on this work which has established the prevalence of seeds in GitHub projects, their multiple categories of seed variants, uniqueness and practical useful potential of meta-maintenance, there are many open avenues and challenges for future work: understanding how to manage all seed variants in seed families, further studies of what are useful changes, and tool support to extract specific needs of a seed family to query other repositories, to name a few.

IX. DATA AVAILABILITY

Our online appendix contains the list of the studied 32,007 repositories on GitHub, the list of the targeted 401,610,677 files, the results of the qualitative and quantitative analyses, and survey material. The appendix is available at https://github.com/NAIST-SE/MetaMaintenancePotential.

REFERENCES

[1] W. Fenske, J. Meinicke, S. Schulze, S. Schulze, and G. Saake, "Variant-preserving refactorings for migrating cloned products to a product line," in *Proceedings of the 24th IEEE International Conference on Software Analysis, Evolution and Reengineering*, SANER, 2017, pp. 316–326.

[2] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki, "An exploratory study of cloning in industrial software product lines," in *Proceedings of the 17th European Conference on Software Maintenance and Reengineering*, CSMR, 2013, pp. 25–34.

[3] T. Ishio, Y. Sakaguchi, K. Ito, and K. Inoue, "Source file set search for clone-and-own reuse analysis," in *Proceedings of the 14th International Conference on Mining Software Repositories*, MSR, 2017, pp. 257–268.

[4] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Sci. Comput. Program.*, vol. 74, no. 7, pp. 470–495, May 2009.

[5] J. Ossher, H. Sajnani, and C. Lopes, "File cloning in open source java projects: The good, the bad, and the ugly," in *Proceedings of the 27th IEEE International Conference on Software Maintenance*, ICSM, 2011, pp. 283–292.

[6] R. Koschke and S. Bazrafshan, "Software-clone rates in open-source programs written in c or c++," in *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering*, SANER, vol. 3, 2016, pp. 1–7.

[7] C. V. Lopes, P. Maj, P. Martins, V. Saini, D. Yang, J. Zitny, H. Sajnani, and J. Vitek, "Déjàvu: A map of code duplicates on github," *Proceedings of the ACM on Programming Languages*, vol. 1, OOPSLA, pp. 84:1–84:28, 2017.

[8] M. Gharehyazie, B. Ray, and V. Filkov, "Some from here, some from there: Cross-project code reuse in github," in *Proceedings of the 14th International Conference on Mining Software Repositories*, MSR, 2017, pp. 291–301.

[9] B. Ray and M. Kim, "A case study of cross-system porting in forked projects," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE, 2012, pp. 53:1–53:11.

[10] G. Gousios, M. Pinzger, and A. v. Deursen, "An exploratory study of the pull-based software development model," in *Proceedings of the 36th International Conference on Software Engineering*, ICSE, 2014, pp. 345–355.

[11] G. Gousios, M.-A. Storey, and A. Bacchelli, "Work practices and challenges in pull-based development: The contributor's perspective," in *Proceedings of the 38th International Conference on Software Engineering*, ICSE, 2016, pp. 285–296.

[12] S. Zhou, c. Stănciulescu, O. Leßenich, Y. Xiong, A. Wąsowski, and C. Kästner, "Identifying features in forks," in *Proceedings of the 40th International Conference on Software Engineering*, ICSE, 2018, pp. 105–116.

[13] S. Zhou, B. Vasilescu, and C. Kästner, "What the fork: A study of inefficient and efficient forking practices in social coding," in *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE , 2019, pp. 350–361.

[14] R. Potvin and J. Levenberg, "Why google stores billions of lines of code in a single repository," *Commun. ACM*, vol. 59, no. 7, pp. 78–87, Jun. 2016.

[15] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan, "Lessons from building static analysis tools at google," *Communications of the ACM*, vol. 61, no. 4, pp. 58–66, Mar. 2018.

[16] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: a qualitative approach," *Science of Computer Programming*, vol. 74, no. 7, pp. 470–495, May 2009.

[17] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, Aug. 2002.

[18] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: Scalable and accurate tree-based detection of code clones," in *Proceedings of the 29th International Conference on Software Engineering*, ICSE, 2007, pp. 96–105.

[19] H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, "Accurate and efficient structural characteristic feature extraction for clone detection," in *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering*, FASE, 2009, pp. 440–455.

[20] Y. Sasaki, T. Yamamoto, Y. Hayase, and K. Inoue, "Finding file clones in FreeBSD Ports Collection," in *Proceedings of the 7th IEEE Working Conference on Mining Software Repositories*, MSR, 2010, pp. 102–105.

[21] J. R. Cordy and C. K. Roy, "The nicad clone detector," in *Proceedings of the 19th IEEE International Conference on Program Comprehension*, ICPC, 2011, pp. 219–220.

[22] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "SourcererCC: Scaling code clone detection to big-code," in *Proceedings of the 38th International Conference on Software Engineering*, ICSE, 2016, pp. 1157–1168.

[23] A. Hemel and R. Koschke, "Reverse Engineering Variability in Source Code Using Clone Detection: A Case Study for Linux Variants of Consumer Electronic Devices," in *Proceedings of the 19th IEEE Working Conference on Reverse Engineering*, WCRE, 2012, pp. 357–366.

[24] R. Koschke and S. Bazrafshan, "Software-clone rates in open-source programs written in C or C++," in *Proceedings of the 10th International Workshop on Software Clones*, IWSC, 2016, pp. 1–7.

[25] Y. Dang, D. Zhang, S. Ge, R. Huang, C. Chu, and T. Xie, "Transferring code-clone detection and analysis to practice," in *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*, ICSE-SEIP, 2017, pp. 53–62.

[26] J. Rubin, K. Czarnecki, and M. Chechik, "Managing cloned variants: A framework and experience," in *Proceedings of the 17th International Software Product Line Conference*, SPLC, 2013, pp. 101–110.

[27] V. Bauer and B. Hauptmann, "Assessing Cross-Project Clones for Reuse Optimization," in *Proceedings of the International Workshop on Software Clones*, IWSC, 2013, pp. 60–61.

[28] T. Ishihara, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto, "Inter-project functional clone detection toward building libraries – an empirical study on 13,000 projects," in *Proceedings of the 19th Working Conference on Reverse Engineering*, WCRE, 2012, pp. 387–391.

[29] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE, 2014, pp. 389–400.

[30] K. Chen, P. Liu, and Y. Zhang, "Achieving Accuracy and Scalability Simultaneously in Detecting Application Clones on Android Markets," in *Proceedings of the 36th International Conference on Software Engineering*, ICSE, 2014, pp. 175–186.

[31] J. Davies, D. M. German, M. W. Godfrey, and A. Hindle, "Software bertillonage: Finding the provenance of an entity," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR, 2011, pp. 183–192.

[32] ——, "Software bertillonage: Determining the provenance of software development artifacts," *Empirical Software Engineering*, vol. 18, pp. 1195–1237, Dec. 2013.

[33] M. W. Godfrey and L. Zou, "Using Origin Analysis to Detect Merging and Splitting of Source Code Entities," *IEEE Transactions on Software Engineering*, vol. 31, no. 2, pp. 166–181, Feb. 2005.

[34] D. Steidl, B. Hummel, and E. Juergens, "Incremental Origin Analysis of Source Code Files," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR, 2014, pp. 42–51.

[35] N. Kawamitsu, T. Ishio, T. Kanda, R. G. Kula, C. De Roover, and K. Inoue, "Identifying source code reuse across repositories using LCS-based source code similarity," in *Proceedings of the 14th International Working Conference on Source Code Analysis and Manipulation*, SCAM, 2014, pp. 305–314.

[36] D. Spinellis, "A repository of unix history and evolution," *Empirical Software Engineering*, vol. 22, no. 3, pp. 1372–1404, Aug. 2016.

[37] D. M. German, M. D. Penta, Y.-G. Guéhéneuc, and G. Antoniol, "Code siblings: Technical and legal implications of copying code between applications," in *Proceedings of the 6th Working Conference on Mining Software Repositories*, MSR, 2009, pp. 81–90.

[38] J. Krinke, N. Gold, Y. Jia, and D. Binkley, "Distinguishing copies from originals in software clones," in *Proceedings of the 4th International Workshop on Software Clones*, IWSC, 2010, pp. 41–48.

[39] ——, "Cloning and copying between GNOME projects," in *Proceedings of the 7th IEEE Working Conference on Mining Software Repositories*, MSR, 2010, pp. 98–101.

[40] S. Stanciulescu, S. Schulze, and A. Wasowski, "Forked and integrated variants in an open-source firmware project," in *Proceedings of the 31st IEEE International Conference on Software Maintenance and Evolution*, ICSME, 2015, pp. 151–160.

[45] S. Chacon and B. Straub, *Pro Git*, 2nd ed. Berkely, CA, USA: Apress, 2014.

[41] L. Ren, S. Zhou, C. Kästner, and A. Wąsowski, "Identifying redundancies in fork-based development," in *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering*, SANER, 2019, pp. 230–241.

[42] D. Strüber, M. Mukelabai, J. Krüger, S. Fischer, L. Linsbauer, J. Martinez, and T. Berger, "Facing the truth: Benchmarking the techniques for the evolution of variant-rich systems [research]," in *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume A*, SPLC, 2019, pp. 26:1–26:12.

[43] J. Jang, A. Agrawal, and D. Brumley, "ReDeBug: Finding unpatched code clones in entire OS distributions," in *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, SP, 2012, pp. 48–62.

[44] H. Li, H. Kwon, J. Kwon, and H. Lee, "CLORIFI: software vulnerability discovery using code clone verification," *Concurrency and Computation: Practice and Experience*, vol. 28, no. 6, pp. 1900–1917, Apr. 2016.

[46] H. Hata, C. Treude, R. G. Kula, and T. Ishio, "9.6 million links in source code comments: Purpose, evolution, and decay," in *Proceedings of the 41st International Conference on Software Engineering*, ICSE, 2019, pp. 1211–1221.

[47] A. La, "Language Trends on GitHub – The GitHub Blog," https://blog.github.com/2015-08-19-language-trends-on-github/, 2015, [Online; accessed Aug 2018].

[48] F. Beuke, "Github Language Statistics – GitHut 2.0," https://madnight.github.io/githut/, [Online; accessed Aug 2019].

[49] GitHub, "The State of the Octoverse 2018," https://octoverse.github.com/, 2018, [Online; accessed Aug 2019].

[50] G. Gousios, "The ghtorent dataset and tool suite," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR, 2013, pp. 233–236.

[51] M. Aniche, G. Bavota, C. Treude, M. A. Gerosa, and A. Deursen, "Code smells for model-view-controller architectures," *Empirical Software Engineering*, vol. 23, no. 4, pp. 2121–2157, Aug. 2018.

[52] J. L. C. Izquierdo, V. Cosentino, and J. Cabot, "An empirical study on the maturity of the eclipse modeling ecosystem," in *Proceedings of the ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems*, MODELS, 2017, pp. 292–302.

[53] J. Coelho and M. T. Valente, "Why modern open source projects fail," in *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE, 2017, pp. 186–196.

[54] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan, "Curating GitHub for engineered software projects," *Empirical Software Engineering*, vol. 22, no. 6, pp. 3219–3253, Dec. 2017.

[55] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies?" *Empirical Software Engineering*, vol. 23, no. 1, pp. 384–417, Feb. 2018.

[56] L. Ren, "Automated patch porting across forked projects," in *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE, 2019, pp. 1199–1201.

[57] T. Pfofe, T. Thüm, S. Schulze, W. Fenske, and I. Schaefer, "Synchronizing software variants with variantsync," in *Proceedings of the 20th International Systems and Software Product Line Conference*, SPLC, 2016, pp. 329–332.

[58] R. Yue, N. Meng, and Q. Wang, "A characterization study of repeated bug fixes," in *Proceedings of the 33rd IEEE International Conference on Software Maintenance and Evolution*, ICSME, 2017, pp. 422–432.