

Forking Without Clicking: on How to Identify Software Repository Forks

Antoine Pietri, Guillaume Rousseau, Stefano Zacchiroli

▶ **To cite this version:**

Antoine Pietri, Guillaume Rousseau, Stefano Zacchiroli. Forking Without Clicking: on How to Identify Software Repository Forks. MSR 2020 -17th International Conference on Mining Software Repositories, Oct 2020, Seoul, South Korea. pp.25 - 26, 10.1145/3379597.3387450 . hal-02527811v2

HAL Id: hal-02527811

https://inria.hal.science/hal-02527811v2

Submitted on 15 Nov 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Forking Without Clicking: on How to Identify Software Repository Forks

Antoine Pietri
antoine.pietri@inria.fr
Inria
Paris, France

Guillaume Rousseau
guillaume.rousseau@u-paris.fr
Université de Paris and Inria
Paris, France

Stefano Zacchiroli
zack@irif.fr
Université de Paris and Inria
Paris, France

ABSTRACT

The notion of *software "fork"* has been shifting over time from the (negative) phenomenon of community disagreements that result in the creation of separate development lines and ultimately software products, to the (positive) practice of using distributed version control system (VCS) repositories to collaboratively improve a single product without stepping on each others toes. In both cases the VCS repositories participating in a fork share parts of a common development history.

Studies of software forks generally rely on hosting platform metadata, such as GitHub, as the source of truth for what constitutes a fork. These "forge forks" however can only identify as forks repositories that have been created *on* the platform, e.g., by clicking a "fork" button on the platform user interface. The increased diversity in code hosting platforms (e.g., GitLab) and the habits of significant development communities (e.g., the Linux kernel, which is not primarily hosted on any single platform) call into question the reliability of trusting code hosting platforms to identify forks. Doing so might introduce selection and methodological biases in empirical studies.

In this article we explore various definitions of "software forks", trying to capture forking workflows that exist in the real world. We quantify the differences in how many repositories would be identified as forks on GitHub according to the various definitions, confirming that a significant number could be overlooked by only considering forge forks. We study the structure and size of fork networks, observing how they are affected by the proposed definitions and discuss the potential impact on empirical research.

KEYWORDS

software evolution, source code, software fork, open source, free software, version control system

ACM Reference Format:
Antoine Pietri, Guillaume Rousseau, and Stefano Zacchiroli. 2020. Forking Without Clicking: on How to Identify Software Repository Forks. In *17th International Conference on Mining Software Repositories (MSR '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3379597.3387450

Publication rights licensed to ACM. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of a national government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.
MSR 2020, 25-26 May, 2020, Seoul, South Korea
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-7517-7/20/05…$15.00
https://doi.org/10.1145/3379597.3387450

1 INTRODUCTION

How developers and software communities work on their projects, and how this relationship evolves over time, have been topics of interest in software engineering research for many decades.

Historically, *software "forking"* [21] has been intended as the practice of taking the source code and development history of an existing software product to create a new, competing product, whose development will happen elsewhere and taken to different directions. This kind of "hard fork" is enabled by free/open source software (FOSS) licensing [10] and its possibility is an asset that guarantees freedom of development; while the actual occurrence of a hard fork has generally been considered a liability [28] for project sustainability [20, 22, 26].

In the past decade the rise in popularity of *distributed* version control systems (DVCS) [31] introduced a significant shift of paradigm and terminology. The expression "fork" is now generally intended [36] to refer to the mere technical act of creating a new VCS repository that contains the full history (at the time of fork) of a preexisting repository, without an implicit negative connotation (also called "development forks" [10]). Repository forks can be created on social coding platforms [7, 33] with as little as a click on a button. Then, while a forked repository *can* be used to hard fork a project, often it is just a way to work on software improvements that will be eventually sent back to the originating project as pull requests [11] for integration.

Likely as a consequence of the prevalence of social coding platforms, recent literature on forks has focused on a single source of truth to determine what constitutes a fork: metadata provided by code hosting providers, and most notably GitHub. Clicking the fork button on GitHub indeed, in addition to cloning development history into a new repository, also registers a "is forked from" relationship between the new repository and its parent. This relationship forms an ancestry graph that GitHub makes available through its API and that is what has traditionally been studied as a large, easily exploitable fork network.

The first drawback of trusting platform metadata as source of truth for what repository is a fork is that it is platform-specific. One cannot identify as forks repositories hosted on GitHub that has been forked from, say, GitLab, or more generally non-GitHub hosted repositories, and vice-versa. Similarly, although arguably less relevant from a quantitative point of view, one cannot recognize as forks, say, Git repositories used to collaborate with Subversion repositories via `git-svn`. For a fork ecosystem to be properly studied via the current approach, all the parallel development must happen using the same VCS and on the same platform. While the prevalence of Git does not seem to be waning, Git code hosting

diversity is increasing, making the platform-specific part of this problem potentially severe.

A second, more subtle methodological drawback is that trusting platform metadata introduces a selection bias on both the amount and type of forks that are considered. The fact that social coding platform strongly encourage, and sometimes even automate, the creation of forked repositories as the main way to contribute even the smallest one-liner change, inflates the number of forks. Many of these (soft) forks will be short-lived in terms of development activity. Hard forks will comparatively be more long lived and will not necessarily reside on the same code hosting platform. The example of the Linux kernel community is revealing in this respect: several copies of the full development history of Linux exist on GitHub, but are not recognizable as forks of `torvalds/linux` according to platform metadata, because kernel development does not primarily happen on GitHub and kernel developers create their repositories using `git clone`.

Fork inflation also results in increased duplication of software artifacts (source code files or directories, commits, ...) across repositories [29], which has a significant impact on fork studies that rely on metrics as simple as repository size (measured as the number of hosted commits). Filtering out forked repository is a common solution to this problem, which calls into question *how* to properly identify forks.

The absence of extensive, homogeneous fork research has been pointed out in the past as a missing piece [28] in the literature. In this paper we try to provide methodological tools to enable fork studies that do not restrict themselves to platform metadata to recognize forks, thereby removing the constraint of analyzing a single platform and mitigating the risk of selection biases.

As an alternative to relying on platform metadata to recognize forks we propose to compare the content of VCS and consider as forks repositories that share artifacts such as commits or entire source trees. We will explore the impact of different such definitions and compare their impact in terms of the amount and structure of forks identified using platform metadata. Specifically, we will answer the following research questions:

RQ1: how do code hosting platform information about which VCS repositories are forks compare to the presence of shared source code artifacts in repositories?

RQ2: how are (a) the amount of forks and (b) the structure of fork networks affected by fork definitions based on VCS artifact sharing?

RQ1 will intuitively assess the level of trustworthiness of platform fork metadata: if many repositories, e.g., share commits but are not identified as fork by platform metadata, then relying on those metadata alone would appear to be methodologically dangerous. As one might consider different types of shared VCS artifacts (commits, source tree directories, individual files, ...) as fork evidence, RQ2 will provide an empirical evaluation of the effects of basing fork definitions on one or the other.

Paper structure. Section 2 explores the spectrum of fork definitions considered in the paper. Section 3 presents the experimental methodology and used datasets. Results are discussed in Section 4, threats to their validity in Section 5. Before concluding, related work is discussed in Section 6.

Replication package. A replication package for this paper is available from Zenodo at https://zenodo.org/record/3610708.

2 WHAT IS A FORK?

In this section we explore the spectrum of possible definitions of what constitutes a *fork*. In the following we will use the term "fork" to mean a forked software *repository*, without discriminating between "hostile" (or hard forks, according to the terminology of [36]) and development forks. We propose three definitions, corresponding to three types of forks—type 1 to 3, reminiscent of code clone classification [27, 30]—along a spectrum of increased sharing of artifacts commonly found in version control systems (VCS), such as commits and source code directories.

The first definition, of type 1 forks, relies solely on code hosting platform information and requires no explicit VCS artifact sharing between repositories to be considered forks (although it allows it):

Definition 2.1 (Type 1 fork, or forge fork). A repository B hosted on code hosting platform P is a *type 1 fork* (or *forge fork*) of repository A hosted on the same platform, written $A \rightsquigarrow_1 B$, if B has been created with an explicit "fork repository A" action on platform P.

Although informal and seemingly trivial, this definition is both meaningful and actionable on current major code hosting platforms. For example, GitHub stores an explicit "forked from" relationship and makes it available via its repositories API:[1]

> The `parent` and `source` objects are present when the repository is a fork. `parent` is the repository this repository was forked from, `source` is the ultimate source for the network.

GitLab does the same and exposes type 1 fork information via its projects API:[2]

> If the project is a fork, and you provide a valid token to authenticate, the `forked_from_project` field will appear in the response.

which corresponds to exploitable JSON metadata such as:

```
{
   "id":3,
   ...
   "forked_from_project":{
      "id":13083,
      "description":"GitLab Community Edition",
      "name":"GitLab Community Edition",
      ...
      "path":"gitlab-foss",
      "path_with_namespace":"gitlab-org/gitlab-foss",
      "created_at":"2013-09-26T06:02:36.000Z",
   ...
```

Without getting too formal we observe that each repository is the forge fork of at most one repository (its parent) and that the relation of being a forge fork is: not reflexive ($A \not\rightsquigarrow_1 A$), not symmetric ($A \rightsquigarrow_1 B$ does not imply—and, in fact, excludes—that $B \rightsquigarrow_1 A$), not transitive ($A \rightsquigarrow_1 B$ and $B \rightsquigarrow_1 C$ does not imply—and in fact, due to parent uniqueness, excludes—that $A \rightsquigarrow_1 C$).

[1] https://developer.github.com/v3/repos/, retrieved 2020-01-13.
[2] https://docs.gitlab.com/ee/api/projects.html, retrieved 2020-01-13

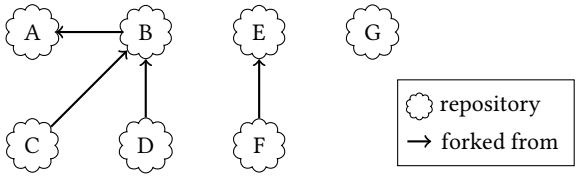


Figure 1: Type 1 forks, or forge forks, as declared on code hosting platforms. Repository B is a forge fork of A, C and D are forge fork of B, F of E, while no repository is a forge fork of G. Note how this definition induces a global, directed, forge fork graph (specifically: a forest of disjoint trees).

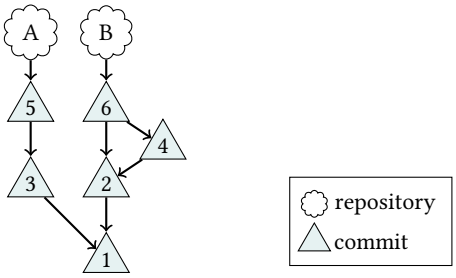


Figure 2: Type 2 forks, or shared commit forks. Repository A is a fork of B and vice versa, since they share commit 1.

The latter might seem surprising at first but is consistent with the definition, because the action resulting on the creation of C happened on B, not A. (We will introduce later a related notion of repository relationship that captures transitivity.)

Forge forks induce a global directed graph on repositories, specifically a forest of disjoint fork-labeled trees, as depicted in Figure 1.

Type 2 forks, or *shared commit forks*, are based on the ability offered by most VCS (and all distributed VCS) of globally identifying commits across any number of repositories, usually by the means of intrinsic commit identifiers based on cryptographic hashes [8, 31]. Given the ability to identify commits across different repositories we can define type 2 forks as follows:

Definition 2.2 (Type 2 fork, or shared commit fork). A repository B is a *type 2 fork* (or *shared commit fork*) of repository A, written $A \rightsquigarrow_2 B$ if it exists a commit c contained in the development histories of both A and B.

Figure 2 shows an example of 2 repositories, A and B that are type 2 forks of each other, due to the fact they have in common commit 1, the initial commit; their respective development histories diverged immediately after that commit and never shared any other commits. In the general case shared commit forks will share many more commits: all the commits that were available at the time of the most recent development history divergence.

Differently from type 1 forks, the relation of being a type 2 fork is symmetric ($A \rightsquigarrow_2 B$ implies $B \rightsquigarrow_2 A$), but still not transitive (as three repositories A, B and C can have shared artifacts between A and B and between B and C without there necessarily being a shared artifact between A and C).

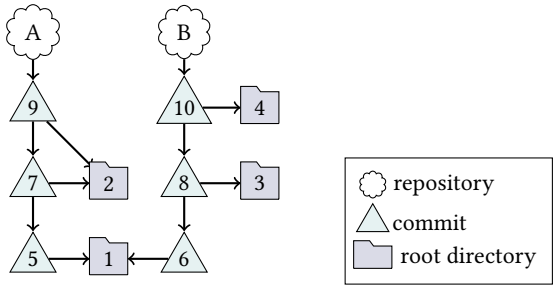


Figure 3: Type 3 fork, or shared root fork. Repository A is a fork of B and vice versa, since they share root directory 1. As per shared commit forks, type shared root forks are symmetric.

Intuitively, the notion of shared commit forks is more robust than that of forge forks because it allows to recognize as forks—in the broad sense of "repositories that collaborate with one another"—repositories that are hosted on different platforms. A repository hosted on GitLab.com, or your personal git repository on your homepage, can be recognized as a fork of a another hosted on GitHub. The price to pay is that, due to symmetry, the definition *alone* is not enough to orientate the relation; it does not capture which repository "came first".

We can push this idea further, trying to make it even more robust, and capable of recognizing as forks repositories that have no recognizable shared commits, but do share entire source trees. That is of interest when, for example, collaboration happens using different version control systems (e.g., a developer using git-svn to participate into the development of a Subversion based project). Type 3 forks, or *shared root (directory) forks*, allow to capture those situations:

Definition 2.3 (Type 3 fork, or shared root fork). A repository B is a *type 3 fork* (or *shared root fork*) of a repository A, written $A \rightsquigarrow_3 B$, if there exist a commit c_A in the development history of A and a commit c_B in that of B such that the full source code trees of the two commits are identical.

The intuition behind type 3 forks is depicted in Figure 3. Note that it is not enough for the two repositories to share any arbitrary *sub*-directory to be considered forks, as that would consider as forks repositories that embed third-party libraries, an arguably undesired consequence; we need the *root* directories of two commits to be (recursively) equal for establishing a shared root fork relationship.

The same properties of type 2 forks apply to type 3 forks: the shared root fork relation is also symmetric. In most VCS, and in all modern DVCS, type 2 forks is also a strictly larger relation than type 3 forks: $A \rightsquigarrow_2 B$ implies $A \rightsquigarrow_3 B$, because if there exists a shared commit c that makes A and B shared commit forks, then the root directory pointed by c also makes A and B shared root forks (due to the cryptographic properties of intrinsic commit identifiers in DVCS). This property of inclusion, in the sense of one definition implying the other, is at the heart of the analysis made in section 4.3, studying the aggregation processes of networks and cliques.

In theory we could go further, and introduce an even more lax notion of fork, that equates repositories sharing as little as a single file, but that would exacerbate the problematic behavior we discussed for sharing sub-directories.

Armed with these definitions we will be able to answer RQ1, by comparing the number of forks identified by Definition 2.1 with those identified by Definition 2.2 and/or 2.3 (that we refer to as *intrinsic* forks). To fully address RQ2 on the other hand we need to capture the notion of "community" of repositories used for collaboration, as follows:

Definition 2.4 (Type T fork network). The *type T fork network* of a repository A is the smallest set $\mathcal{N}_A^T$ such that:

- $A \in \mathcal{N}_A^T$
- $\forall B \in \mathcal{N}_A^T,\ B \rightsquigarrow_T C \implies C \in \mathcal{N}_A^T$
- $\forall B \in \mathcal{N}_A^T,\ C \rightsquigarrow_T B \implies C \in \mathcal{N}_A^T$

That is, a fork network is the set of all repositories reachable from a given one, following both forked from (parents) and forked to repositories (children). The definition is parametric in the type of forks, so we have type 1 fork networks ($\mathcal{N}^1$), type 2 fork networks ($\mathcal{N}^2$), and type 3 fork networks ($\mathcal{N}^3$).

A stricter notion that will come in handy is that of repository cliques, sets of repositories that are all direct forks (i.e., neither transitive nor reverse transitive) of each other:

Definition 2.5 (Type T fork clique). The *type T fork clique* of a repository A is the largest set C_A^T such that:

- $A \in C_A^T$
- $\forall C, (\forall B \in C_A^T, B \rightsquigarrow_T C \wedge C \rightsquigarrow_T B) \implies C \in C_A^T$

Note that, while this definition is parametric in the type of forks too, fork cliques make intuitive sense only for type 2 and type 3 forks; type 1 forks (forge forks) only have singleton cliques as the relation is not symmetric.

3 METHODOLOGY

3.1 Dataset

Our goal is to experimentally determine the amount and structure of forks for the various definitions we have introduced. To do so we will use two datasets: the Software Heritage Graph Dataset [25], which contains the development history needed to find intrinsic fork relationships, and a reference forge-specific dataset, GHTorrent [12], which contains the fork ancestry relationships as captured by GitHub.

GHTorrent. GitHub is the largest public software forge, and is therefore the candidate of choice to study forge forks (type 1). GHTorrent [12] crawls and archives GitHub via its REST API and makes periodical data dumps available in a relational table format. In its database schema, the `project` table contains a unique identifier for each repository, and a `forked_from` column contains the ID of the repository it has been forked from if the repository is considered to be a forge forks. A single SQL query on this table allows to extract the full graph of GitHub-declared forks, e.g.:[3]

[3] Additional URL gymnastic is needed in the query to cross-reference GHTorrent project URLs with Software Heritage ones; we refer to the replication package for this kind of details.

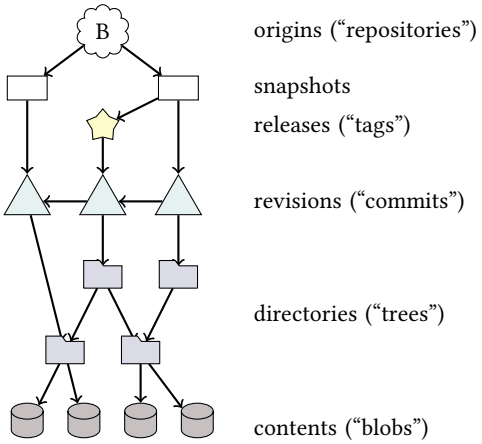


Figure 4: The Software Heritage Graph Dataset data model

```
select parents.url as parent,
       projects.url as child
from projects
inner join projects as parents
      on projects.forked_from = parents.id
```

Software Heritage Graph Dataset. Software Heritage [1, 9] is the largest publicly accessible archive of software source code and accompanying development history, spanning more than 90 million software projects retrieved from major development forges including GitHub and GitLab.com. The Software Heritage Graph Dataset [25] is an offline dataset containing the development history of all the projects in Software Heritage in a tabular representation of a unified directed acyclic graph (DAG). As the archive encompasses a substantial portion of all the public GitHub repositories, it is possible to cross-reference the origins contained in this dataset with the ones in GitHub, our reference for forge forks.

The Software Heritage Graph Dataset data model maps the traditional concepts of VCS as nodes in a Merkle DAG [19], as shown in Figure 4. As a consequence, all the development artifacts, including commits and source trees, are natively deduplicated within and across projects. This property is particularly useful to find intrinsic forks, as it enables tracking the relevant artifacts (revisions and directories) across the entire dataset and link them back to their source repositories.

The dataset contains two intermediate layers between repositories and the commit graph they point to: *snapshots*, which are point in time captures of the state of a repository; and *releases* (or "tags"), which are *revisions* (or commits) labeled with a specific name. As none of our fork definitions depend on these artifacts, the two layers can be flattened out so that the origins point directly to the revision graph. Likewise, the blob layer and the directory layer are not needed to find shared commit forks (Definition 2.2), while shared root forks (Definition 2.3) only require the root directory of each revision. Filtering out the unnecessary nodes reduces the graph to a more reasonable size of 2 billion nodes (down from 10 billion), which makes it easier to process on a single machine. The structure of the resulting subgraph closely matches the examples in Figure 2 and Figure 3, making it easy to verify the intrinsic definitions.

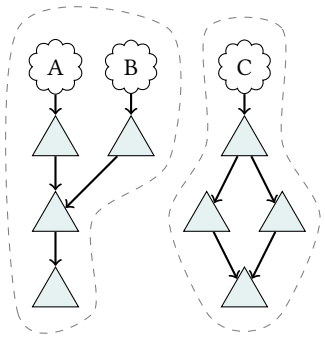


Figure 5: Fork networks identified as connected components, for the case of shared commit (type 2) forks. Connected components are computed on the undirected version of the shown Merkle DAG. Measuring network sizes as the number of contained repository nodes we obtain that: repositories A and B are forks of each other and members of a network of size 2, while repository C is in its own singleton network.

We run the experiments on the compressed version of the two graph datasets, using the WebGraph framework [4, 5]. The Software Heritage Graph Dataset is already distributed as a compressed BVGraph, along with the `swh-graph` helper library [3] to run traversal algorithms easily. The GHTorrent can be compressed from its relational database format using the `swh-graph compress` utility.

3.2 Fork networks

The easiest way to get a first sense of the amount and structure of forks according to the various definitions is to find all fork networks, as per Definition 2.4. This can be done in linear time with a simple graph traversal with linear complexity: two repositories are in the same network if and only if there exists a path between them in the undirected subgraph of origins and revisions. (We recall from the dataset section that we have removed the snapshot and revision layers, so that root commits are directly pointed by repository nodes.) Finding all the fork networks is therefore equivalent to computing the connected components on this subgraph, as exemplified in Figure 5.

Using fork networks has the advantage of allowing easy interpretations of the results. First, it is trivial to quantify how many repositories are forks by counting the number of repositories that belong to non-singleton networks. Besides, a direct comparison can be made between the distribution of forge forks and shared commit or root forks, as networks provide a partition method for both graphs. The sizes of the networks can be directly compared between the three definitions while keeping the invariant of number of total repositories. This is not the case when looking at fork cliques, since the same repository can be found in multiple cliques, which makes comparison harder.

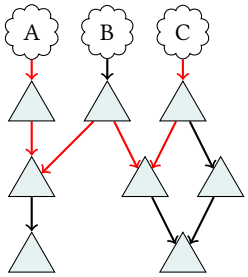


Figure 6: Example of misleading clustering of fork networks. Here, repositories A and C are in the same network because there is a path between them, even though they do not share common development history.

In GHTorrent origins are already linked together in a global graph where the edges represent the forge-level forking relationships. We can partition this forge fork graph in fork networks similarly by computing all its connected components.

Our experimental design is therefore as follows: first, we list the common non-empty repositories between the Software Heritage Graph Dataset and GHTorrent. We then extract the aforementioned subgraphs: the development history graph for Software Heritage (origins → {revisions, releases} → commits) and the fork graph for GHTorrent (origins → origins). We then compute the connected components of each graph using a simple depth-first traversal algorithm, then output the origins contained in each component.

3.3 Fork cliques

While partitioning the corpus in fork networks gives a good idea of how intrinsic forks are linked together, it can group together repositories that are not forks of each other, as the intrinsic fork relationship is not transitive. Figure 6 shows a pattern, that we have verified as commonly found in the wild, where two different cliques will be merged in the same fork network—A and B are part of the same clique as they share development history; the same applies to B and C; whereas A and C do not share any part of their respective development histories but will end up in the same network. We expect this effect to merge cliques into giant components, that will make the size of the largest networks hard to interpret.

The other interesting metric that can be looked at is the distribution of fork cliques, as defined in Definition 2.5. While cliques do not provide a partition function for the graph, they allow to narrow down the actual extent of forking relationships within large fork networks.

Due to the fact that shared commit fork cliques are defined pairwise, the naive algorithm to find all the inclusion-maximal cliques is superlinear: for each repository, walk through its commit history and add all the commits to a queue, then take the transposed graph to walk through the commit history backwards and list all repository leaves. The time complexity of this algorithm is highly unpractical: in the worst case, if all the repositories are forks of each other, is has time complexity of $O(R \times C)$ where C is the number of commits and R the number of repositories in the graph.

However, clever use of some properties on the DAG structure of the commit graph can substantially speed up the algorithm. First,

Algorithm 1 Find all the fork cliques

```
function FindOriginLeaves(r)
    S_O ← empty set
    for all n ∈ AncestorsDFS(r) do
        if type(n) = ORIGIN then
            add n to S_O
        end if
    end for
    return S_O
end function
function FindCliques(G)
    S_F ← empty set
    S_C ← empty set
    for all n ∈ G do
        if type(n) = REVISION and n has no parents then
            c ← FindOriginLeaves(n)
            f_c ← Fingerprint(c)
            if f_c ∉ S_F then
                add f_c to S_F
                add c to S_C
            end if
        end if
    end for
    return S_C
end function
```

Algorithm 2 Compute the p-cliques partition function

```
function CliquesToPartition(L_C)
    ReverseSizeSort(L_C)                ▷ Process larger cliques first
    I ← empty map                       ▷ Build reverse index
    for all c_i ∈ L_C do
        add {i → c_i} to I
    end for
    for all c_i ∈ L_C do
        for all r_j ∈ c_i do
            for all s ∈ I[r_j] do       ▷ Remove subsequent
                if k > i then             occurrences of r_j
                    remove r_j from s
                end if
            end for
        end for
    end for
    L_C ← RemoveEmptySets(L_C)          ▷ Remove cliques left empty
    return L_C
end function
```

fork cliques can be generated by iterating on the common ancestors instead of the repositories: for each commit c, if it has more than one repository leave when doing a traversal on the transposed graph, then c was a common commit ancestor, and the generated set of repositories is a fork clique. Besides, since the ancestry relationship is transitive, the clique with commit c as a common ancestor is the same as the clique generated by running the traversal on its parents. By induction, it is possible to compute all the cliques simply by doing one traversal per "root" commit.

The resulting algorithm is Algorithm 1: for each root commit with no parents, we generate the clique of all repositories that contain it. We use a cryptographic hash fingerprint to avoid adding multiple times the same clique if it has multiple root commits. While this algorithm technically does not change the worst case complexity on arbitrary graphs, it is still a huge speed improvement in our case, as commit chains tend to be degenerate (i.e., very long chains with indegrees and outdegrees close to 1 on average). Algorithm 1 has a best-case complexity of $\Theta(C)$, equivalent to a single DFS traversal. The commit graph is largely close to this best-case scenario, making the algorithm run in just a few hours on the entire corpus.

While Algorithm 1 works well for shared commit forks, the speedup does not apply to shared root forks: the induction property no longer works for root directories, as they are not organised in nearly-degenerate chains. The time complexity for type 3 forks is closer to the worst case of $O(C \times R)$, which makes the clique analysis impractical for this kind of forks.

P-clique partition function. While cliques do not directly provide a way to partition the corpus in several fork clusters (because a single origin can be contained in multiple cliques), it is possible to define a partition function based on them, e.g., by always assigning repositories to the largest clique they belong to. As repositories belonging to multiple cliques appear to be a quite rare occurrence (as they require the equivalent of a `git merge --allow-unrelated-histories` on two completely different repositories), the arbitrary criterion choice is not expected to be a significant caveat to interpret the results.

We use the Algorithm 2 to generate the partition function of the graph based on cliques. To implement the criterion of attributing repositories to their largest cliques, cliques are processed in decreasing order of size. Building a reverse index of "repository → clique it belongs to" allows direct access to the subsequent occurrences of repositories in smaller cliques to remove them. After doing so, the cliques left empty are removed and the newly generated graph partition can be returned.

The output of this algorithm generate a set of sets of origins that are subsets of the input fork cliques. We call this set *"fork p-cliques"* to emphasize the fact that they form a partition of the repository set in which all the groups are fork cliques.

Once this p-clique graph partition is established, the fork definition can once again be compared with the forge definition, by looking at the difference between the size distribution of the partitioned cliques of type 2 forks and the size distribution of networks for type 1 forks.

4 RESULTS

We identified 71.9 M repositories in common between the Software Heritage Graph Dataset and GHTorrent, 41.4 M of which are non-empty. We focused our experiments on these repositories.

Table 1: Number of forks and networks by fork type

Fork type	# forks	# networks
Forge forks (type 1)	18.5 M (44.7%)	25.3 M
Shared commit forks (type 2)	20.1 M (48.4%)	24.0 M
Shared root forks (type 3)	25.3 M (61.1%)	18.5 M

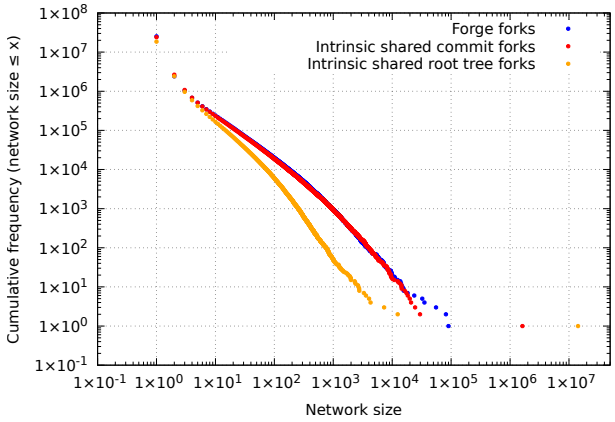


Figure 7: Cumulative frequency distribution of fork network sizes

4.1 Fork networks

In the GHTorrent graph, we found 25.3 M different connected components, among which 22.9 M repositories isolated in their own component, which means they are not forge forks (type 1) of other repositories. The other 2.4 M connected components contain the remaining 18.5 M repositories, which are all in fork networks. These forge forks represent 44.74% of all repositories.

In the Software Heritage Graph Dataset, we found 24.0 M connected components, among which 21.3 M isolated repositories. The remaining 2.6 M components contain 20.1 M shared commit forks (type 2), i.e., 48.44% of all repositories. We have hence almost 9% *more* shared commit forks than forge forks, which is a significant divergence for the most strict definition of forks based on shared VCS artifacts.

For shared root forks (type 3), we found 18.5 M connected components, among which 16.1 M isolated repositories and 25.3 intrinsic forks (61.08% of all repositories), which is almost 37% more than the forge forks. These results are summarized in Table 1. They suggest that **in between 1.6 M (3.8% of total) and 6.8 M (16%) repositories might be overlooked when studying forks using only GitHub metadata** as a source of truth for what is a fork.

Figure 7 shows the cumulative frequency distribution of fork networks for intrinsic forks and forge forks. That is, for each fork network size x, the number of repositories in networks of size $\geq x$ is shown. At first glance, the distribution of forge forks and shared commit forks appear to be pretty similar (although the log scale minimizes the differences between the two), which is a good sign that the two definitions are not returning vastly different results. The average size of fork networks is also about the same ($\approx$ 7.6 for type 2 forks, $\approx$ 7.7 for type 1). The situation appears to be quite different for shared root forks, where the average size is $\approx$ 10.5 and the frequency distribution is significantly farther from the reference distribution of forge forks.

One distinguishing feature of each distribution of type 2 and type 3 forks is the size of the largest connected component, which is significantly larger than the largest networks of forge forks (by a factor of 17 for shared revision forks, and 157 for shared root forks). As discussed in Section 3.3, this is an expected outcome of our use of network as a quantification metric and confirms the need for further analysis through fork cliques. This does not however have any implications on the quantification aspect of the experiment, as partitioning this network further using fork cliques would still yield the same number of non-isolated repositories.

4.2 Fork cliques

As expected, running Algorithm 1 to generate the shared-revision cliques on the compressed graph does not take more than an hour, which is the same order of magnitude as the time needed for a simple full traversal of the revision graph [3]. This confirms our prediction that in the shared-revision case, the average-case runtime of the algorithm is close to $\Theta(R)$.

The algorithm finds 24.5 M cliques, although the results are difficult to interpret in this current state as the cliques overlap together. A few key observations can nevertheless already be made, notably the absence of very large cliques: the largest clique contains 92.4 M repositories, which is very similar to the largest forge fork network (which contains 90.2 M repositories). This is consistent with our intuition expressed in Section 3.3 that the largest intrinsic fork networks are a specific feature of networks (as seen in Figure 6), and that these artifacts disappear when looking at the cliques. It is also possible to measure how the cliques overlap: 28 M repositories are present in a single clique, while the remaining 13.3 M appear two times or more. On average, each repository appears in $\approx$ 1.47 cliques.

Computing the p-clique partition function using Algorithm 2 removes this overlap to allow a direct comparison with the forge fork networks. This algorithm takes a few minutes to process the 24 million cliques and returns the p-clique partition directly, restoring the invariant of total number of repositories (41.5 M).

There are 24.0 M of p-cliques partitioning the graph, which is pretty close to the number of forge fork networks in GitHub (25.3 M). 21.3 M repositories are isolated in their own p-clique (51.6%), and the remaining 48.4% are in cliques of size larger than one, which is consistent with the findings of Section 4.1 which uses fork networks as a quantification mechanism.

Figure 8 shows the cumulative frequency distribution of the sizes of the shared-commit fork cliques, compared to the baseline of forge fork networks. As before, the graph can be read as: "for each clique (resp. forge fork network) of size x, the number of repositories found in cliques (resp. networks) of size $\geq x$".

The visual similarity between the two distributions is striking: while the shared-commit p-clique distribution seems to be consistently above the forge-fork network baseline for groups of size ≥ 2, they always appear to be very close to each other, even farther in the tail. This **suggests that type 2 forks capture well what developers typically recognize as forks**.

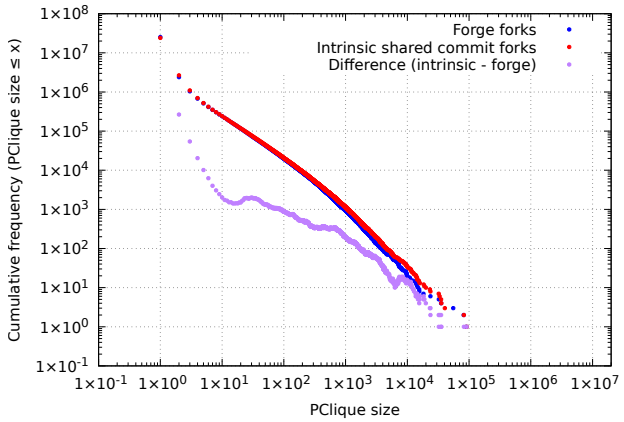


Figure 8: Cumulative frequency distribution of intrinsic fork p-cliques compared to forge fork networks

To formally assess this similarity, the graph also exhibits the cumulative difference between the clique distribution and the baseline. This is in essence, the cumulative size distribution of the cliques of forks *overlooked* when using only the GitHub metadata. This cumulative distribution **mostly stays positive, suggesting that using DVCS data to identify forks is overall a net gain in coverage**. It also appears that the difference is typically at least one order of magnitude less than the size of the clusters, emphasizing the proximity between the two definitions.

4.3 Aggregation process

Two repositories having a common commit ancestor necessarily have a common root source tree (the root source tree of that common commit ancestor), so all the repositories that belong to the same shared-commit network also belong to the same shared-directory network. Similarly, we expect that most origins declared as forge forks will be in the same shared commit and shared root source tree fork networks. By switching from one definition to another, we expect the clusters to aggregate together smaller clusters from the previous definitions.

To characterize this aggregation process into fork clusters at different granularities, we compute the Kolmogorov-Smirnov (KS) distance between the weighted cumulative distributions function of the clique or network size.

We note δO the KS difference between a fork definition A and a fork definition B, and represent it as a function of the size of the network (or partitioned clique). By definition δO is always equal to zero for sizes $s = 1$ (as all the forks are in clusters of size $s \geq 1$) and $s = \max(\text{cluster sizes})$ (as there are no clusters larger than this size).

Because the total number of repositories is invariant, we can plot the KS distance weighted by repositories to see how the repositories found in fork networks (or cliques) of a given size will progressively aggregate into fork networks (or cliques) of different sizes. Figure 9 represents δO between the forge fork definition baseline and: shared commit fork networks (top), shared commit p-cliques (bottom), and shared root tree fork networks (middle)).

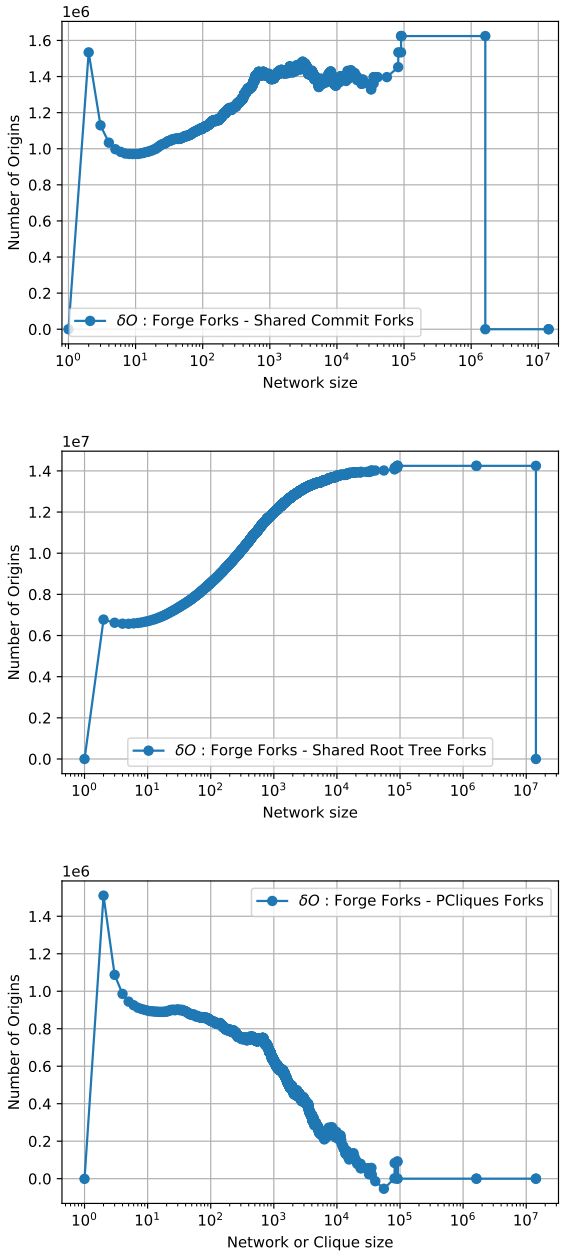


Figure 9: Complementary Cumulative Weighted Distribution Functions Differences between forge fork network and shared-commit fork network (top), shared root source tree fork network (middle), and p-cliques based fork network (bottom).

While this analysis shows the flux of repositories between clusters identified by the different definitions, it can mask some compensating phenomena by merging independent processes, as some repositories can migrate from larger to smaller clusters, sometimes leading to $\delta O < 0$ (Figure 9, bottom, $size \sim 10^5$).

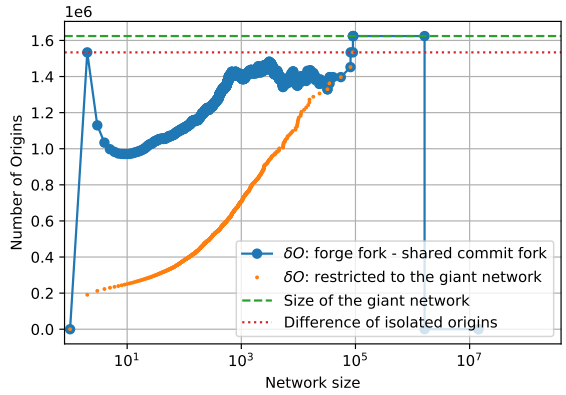


Figure 10: Contribution (orange dots) of the giant (largest) network that appear using shared commit fork definition w.r.t. δO : forge fork − shared commit fork (Same as Fig. 9, Top)

To narrow down this phenomenon, we specifically focus on the largest shared-commit fork network to see how it contributes to the global flux. By taking the repositories in this network and the size distribution of the forge fork networks, we show in Figure 10 the repository flux, as defined above, and compare it to the corresponding global flux.

Several points are noteworthy. First, only 15% (200 k origins over 1.53 M (red dotted)) of the origins that were isolated (blue dot for $size = 2$) in forge fork network are aggregated in the giant network of shared commit fork. This shows that the aggregation mechanism is not only to this "giant" network, since $\sim$ 85% of the origins are aggregated within smaller networks.

Then, the flux for the 6 largest networks (isolated blue and orange dots around 10^5 that overlap) is almost the same whether we restrict ourselves to the origins of the giant network (orange line) or to all the networks (blue line). We conclude that aggregation for the large network sizes is dominated by absorption into the giant cluster, without any redistribution to smaller networks.

This confirms that the "aggregation/merge" mechanism which happens when changing the definition is not just an absorption phenomenon into a "super attractor", but concerns all network sizes, with larger networks absorbing networks of any size.

5 THREATS TO VALIDITY

Internal validity. Aside from forge forks (type 1) we have no certainty on how well the proposed fork definitions capture what developers would recognize as forks. While shared commit (type 2) and shared root (type 3) forks make intuitive sense, the sheer volume of data to be analyzed makes it very hard to rule out the existence of pathological cases. Certainly type 2 and type 3 fork definitions can be "gamed", making unrelated repositories appear as forks when they intuitively are not. Unusual development workflows might also induce topology in the global development graph that merge together repositories that would not be considered forks by developers. There exists an apparent trade-off here between fully automatable definitions based on VCS artifact sharing, and qualitative assessment by developers that does not scale to datasets like the one studied here.

Also as a consequence of the above we do not feel confident at this stage in making a methodological judgment call on whether type 2 or type 3 fork definitions "better" capture the essence of a fork. We simply warn scholars about the extent of the discrepancies between the number of forks detectable via shared VCS artifacts and forge-level metadata. Further work, of both statistical nature (looking for outliers) and based on structured interviews with developers (to review uncommon cases), is needed to improve over this point.

External validity. The datasets used in this study do not capture the full extent of publicly available development history, notably due to their snapshot nature and their assembly through periodic crawling processes. The Software Heritage Graph Dataset only contains data from various forges to the extent of what is covered by the Software Heritage archive, which might lag behind the tracked forges, and GHTorrent is GitHub-specific. As we need comparable samples we were limited by the intersection of the two datasets in this study, which composes these limitations. Still, to the best of our knowledge this is one of the largest quantitative fork study to date, having considered more than 40 M public version control system repositories.

In the future it would be interesting to extend this approach to forges that are raising in popularity, and most notably GitLab. For that we would need a GHTorrent equivalent (or corresponding ad hoc crawling of that forge for the purposes of the study only).

6 RELATED WORK

Accounts of the history of forking have been given by Nyman [21] and Zhou et al. [36, Section 2]. The latter also covers the terminological and cultural shift from hard forks (to be avoided) to forking as the mere technical act of duplicating VCS history, possibly as basis for future collaboration. The present work is agnostic to which interpretation prevails, as in both cases the main observable effect of forking are VCS repositories that share parts of an initially common development history.

Hard forks. Hard forks have been studied extensively in seminal work by Nyman [20–23], covering historical origins, motivations for forking (or not), and sustainability considerations in the socioeconomic context of free/open source software (FOSS) development. Robles and Barahona [28] give a detailed account of famous hard forks, covering history, reasons, and outcomes.

These and other studies of hard forks are qualitative and focused in nature. This paper is complementary to them as it proposes tools to identify and quantitatively measure and observe forks, addressing the need of more extensive and homogeneous fork research already observed in [28]. As far as we could determine without fully replicating the corresponding studies, the VCS repositories involved in the hard fork cases cited thus far would be correctly identified as either type 2 or type 3 forks.

Development forks. With the advent of DVCS and social coding [17], a significant amount of empirical research has been devoted to development forks. Motivations for forking on GitHub have been studied by Jiang et al. [15].

The structure of forks on GitHub has been analyzed in several studies. Thung et al. [33] have characterized the network structure of social coding of GitHub, including forks. Padhye [24] have measured external contributions from non-core developers. Biazzini and Baudry have proposed metrics to quantify and classify collaboration in GitHub repositories pertaining to the same fork tree [2]. Rastogi and Nagappan [26]—as well as Stanciulescu et al. [32] for firmware projects—have characterized forks on GitHub based on the flow of commits between them and the originating repository.

Various performance aspects of the pull request development model [11, 13] have been also studied. Latency in acceptance has been a popular one [34, 35]; the amount of generated community engagement [6, 7] another one. A more general accounting of efficient forking patterns has recently been given by Zhou et al. [36].

To the extent we could determine it without full replication, all aforementioned studies on forking for social coding purposes rely on platform (and more specifically GitHub's) metadata to determine which repository is a fork (of which other). As such, involved repositories would be recognized as type 1 forks, and non type 1 forks (but nonetheless type 2 or 3 forks) might have been overlooked in the studies. To be clear: we have no reason to believe that the findings in those studies would turn out to be different by enlarging the set of considered forks using the alternative definitions. We simply propose to acknowledge fork type discrepancy as an internal validity threat in future studies.

Fork definitions. Aside from the already discussed hard forks v. development forks distinction, the only other work we are aware of on formal or semi-formal fork characterization is [29], which introduces the notion of *most fit fork*: a repository that, within a group of VCS repositories that share commits, contain the largest number of commits. The notion is proposed as a long-term approximation of the main development line of a forked (hardly or otherwise) project. Our notion of type 2 fork clique captures the same idea; additionally we show how to use it to partition the global set of VCS repositories into independent clusters instead of partitioning the global set of commits.

Methodology. Methodological issues and risks in analyzing GitHub were pointed out by Kalliamvakou et al. [16]. While not directly addressed as an explicit risk, forks not recognized as such are echoed by perils *"I: A repository is not necessarily a project"* and *"IX: Many active projects do not conduct all their software development in GitHub"* in that article. Proposed mitigations were, respectively, *"consider the activity in both the base repository and all associated forked repositories"* and *"Avoid projects that have a high number of committers who are not registered GitHub users and projects which explicitly state that they are mirrors in their description"*.

Half a decade later it is arguably *less* of a risk that development happens elsewhere and that a high number of committers are not registered GitHub users (due to the current marked dominance of GitHub). But it is still not zero and might be about to increase again due to push back against centralized services among FOSS developers. In this paper we provide methodological tools and improve upon the mitigation techniques proposed back then. Instead of avoiding projects, one can start from cross-platform datasets [1, 18, 25, 29] and measure the amount of shared VCS artifacts in the available repositories.

7 CONCLUSION

When relying only on forge-specific features and metadata to identify forked repositories, empirical studies on software forks might incur into selection and methodological biases. This is because repository forking can happen exogenously to any specific code hosting platform and out of band, especially when using distributed version control system (DVCS), which are currently very popular among developers.

To mitigate these risks we proposed two different ways to identify software forks solely based on intrinsic VCS data and development history: a *shared commit forks* (type 2) and a *shared root directory forks* (type 3) definition of software forks, as opposed to *forge forks* (type 1) which are identifiable only when created on specific code hosting platforms by, e.g., clicking on a "fork" UI element. We also introduced the notions of *fork cliques* (set of repositories that share parts of a common development history) and *fork networks* (repositories linked together by pairwise fork relationships) as ways to understand and quantify larger sets of forks when using non-transitive definitions of forking.

Via empirical analysis of 40+ M repositories using the GHTorrent and Software Heritage datasets we quantified the amount of type 2 and type 3 forks that are not recognizable as type 1 forks on GitHub, which appears to be substantial: +9% forks for type 2 forks, +37% more for type 3.

We also showed that the aggregation/merge dynamics into larger clusters of related repositories upon changing fork definitions is not just an absorption phenomenon into a "super attractor" cluster, but that it concerns all clusters: smaller ones are absorbed into larger ones of any size.

The methodological implications of our findings are that:

- Empirical software engineering studies on software forks aiming to be exhaustive in their coverage of forked repositories should consider using fork definitions based on *shared VCS history* rather than trusting forge-specific metadata.
- Depending on the research question at hand, the objects of studies to consider when looking at repositories involved in forks are either *fork networks* or *fork cliques*. The latter have the advantage of excluding cases that exist in the wild (e.g., on GitHub) in which repositories that do not share VCS artifacts might end up in the same fork network due to transitiveness.
- Any set of repositories can be partitioned in accordance with its relevant shared commit fork cliques by computing its *fork p-clique* partition function. This way of grouping together repositories that are all type 2 forks of each other is easily substitutable to partition approaches based on forge fork metadata.

ACKNOWLEDGMENTS

The authors would like to thank Théo Zimmermann for his careful review and comments on an early version of this paper.

REFERENCES

[1] Jean-François Abramatic, Roberto Di Cosmo, and Stefano Zacchiroli. 2018. Building the Universal Archive of Source Code. *Commun. ACM* 61, 10 (Sept. 2018), 29–31. https://doi.org/10.1145/3183558

[2] Marco Biazzini and Benoit Baudry. 2014. May the fork be with you: novel metrics to analyze collaboration on GitHub. In *Proceedings of the 5th International Workshop on Emerging Trends in Software Metrics*. ACM, 37–43.

[3] Paolo Boldi, Antoine Pietri, Sebastiano Vigna, and Stefano Zacchiroli. 2020. Ultra-Large-Scale Repository Analysis via Graph Compression. In *SANER 2020: The 27th IEEE International Conference on Software Analysis, Evolution and Reengineering*. IEEE.

[4] Paolo Boldi and Sebastiano Vigna. 2004. The webgraph framework I: compression techniques. In *Proceedings of the 13th international conference on World Wide Web, WWW 2004, New York, NY, USA, May 17-20, 2004*, Stuart I. Feldman, Mike Uretsky, Marc Najork, and Craig E. Wills (Eds.). ACM, 595–602. https://doi.org/10.1145/988672.988752

[5] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework II: Codes For The World-Wide Web. In *2004 Data Compression Conference (DCC 2004), 23-25 March 2004, Snowbird, UT, USA*. IEEE Computer Society, 528. https://doi.org/10.1109/DCC.2004.1281504

[6] Laura Dabbish, Colleen Stuart, Jason Tsay, and James Herbsleb. 2012. Leveraging transparency. *IEEE software* 30, 1 (2012), 37–43.

[7] Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. 2012. Social coding in GitHub: transparency and collaboration in an open software repository. In *Proceedings of the ACM 2012 conference on computer supported cooperative work*. ACM, 1277–1286.

[8] Roberto Di Cosmo, Morane Gruenpeter, and Stefano Zacchiroli. 2018. Identifiers for Digital Objects: the Case of Software Source Code Preservation. In *Proceedings of the 15th International Conference on Digital Preservation, iPRES 2018, Boston, USA*. https://doi.org/10.17605/OSF.IO/KDE56

[9] Roberto Di Cosmo and Stefano Zacchiroli. 2017. Software Heritage: Why and How to Preserve Software Source Code. In *Proceedings of the 14th International Conference on Digital Preservation, iPRES 2017*. https://hal.archives-ouvertes.fr/hal-01590958/

[10] Karl Fogel. 2005. *Producing open source software: How to run a successful free software project*. O'Reilly Media, Inc.

[11] Georgios Gousios, Martin Pinzger, and Arie van Deursen. 2014. An exploratory study of the pull-based software development model. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 345–355.

[12] Georgios Gousios and Diomidis Spinellis. 2012. GHTorrent: Github's data from a firehose. In *9th IEEE Working Conference of Mining Software Repositories, MSR*, Michele Lanza, Massimiliano Di Penta, and Tao Xie (Eds.). IEEE Computer Society, 12–21. https://doi.org/10.1109/MSR.2012.6224294

[13] Georgios Gousios, Andy Zaidman, Margaret-Anne Storey, and Arie Van Deursen. 2015. Work practices and challenges in pull-based development: the integrator's perspective. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 358–368.

[14] Imed Hammouda, Björn Lundell, Tommi Mikkonen, and Walt Scacchi (Eds.). 2012. *Open Source Systems: Long-Term Sustainability - 8th IFIP WG 2.13 International Conference, OSS 2012, Hammamet, Tunisia, September 10-13, 2012. Proceedings*. IFIP Advances in Information and Communication Technology, Vol. 378. Springer. https://doi.org/10.1007/978-3-642-33442-9

[15] Jing Jiang, David Lo, Jiahuan He, Xin Xia, Pavneet Singh Kochhar, and Li Zhang. 2017. Why and how developers fork what from whom in GitHub. *Empirical Software Engineering* 22, 1 (2017), 547–578.

[16] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. 2014. The promises and perils of mining GitHub. In *Proceedings of the 11th working conference on mining software repositories*. ACM, 92–101.

[17] Antonio Lima, Luca Rossi, and Mirco Musolesi. 2014. Coding together at scale: GitHub as a collaborative social network. In *Eighth International AAAI Conference on Weblogs and Social Media*.

[18] Yuxing Ma, Chris Bogart, Sadika Amreen, Russell Zaretzki, and Audris Mockus. 2019. World of code: an infrastructure for mining the universe of open source VCS data. In *Proceedings of the 16th International Conference on Mining Software Repositories*. IEEE Press, 143–154.

[19] Ralph C. Merkle. 1987. A Digital Signature Based on a Conventional Encryption Function. In *Advances in Cryptology - CRYPTO '87, A Conference on the Theory and Applications of Cryptographic Techniques, Santa Barbara, California, USA, August 16-20, 1987, Proceedings (Lecture Notes in Computer Science)*, Carl Pomerance (Ed.), Vol. 293. Springer, 369–378. https://doi.org/10.1007/3-540-48184-2_32

[20] Linus Nyman. 2014. Hackers on Forking. In *Proceedings of The International Symposium on Open Collaboration, OpenSym 2014, Berlin, Germany, August 27 - 29, 2014*, Dirk Riehle, Jesús M. González-Barahona, Gregorio Robles, Kathrin M. Möslein, Ina Schieferdecker, Ulrike Cress, Astrid Wichmann, Brent J. Hecht, and Nicolas Jullien (Eds.). ACM, 6:1–6:10. https://doi.org/10.1145/2641580.2641590

[21] Linus Nyman and Mikael Laakso. 2016. Notes on the History of Fork and Join. *IEEE Annals of the History of Computing* 38, 3 (2016), 84–87. https://doi.org/10.1109/MAHC.2016.34

[22] Linus Nyman and Tommi Mikkonen. 2011. To Fork or Not to Fork: Fork Motivations in SourceForge Projects. *IJOSSP* 3, 3 (2011), 1–9. https://doi.org/10.4018/jossp.2011070101

[23] Linus Nyman, Tommi Mikkonen, Juho Lindman, and Martin Fougère. 2012. Perspectives on Code Forking and Sustainability in Open Source Software, See [14], 274–279. https://doi.org/10.1007/978-3-642-33442-9_21

[24] Rohan Padhye, Senthil Mani, and Vibha Singhal Sinha. 2014. A study of external community contribution to open-source projects on GitHub. In *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 332–335.

[25] Antoine Pietri, Diomidis Spinellis, and Stefano Zacchiroli. 2019. The Software Heritage graph dataset: public software development under one roof. In *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada.*, Margaret-Anne D. Storey, Bram Adams, and Sonia Haiduc (Eds.). IEEE / ACM, 138–142. https://dl.acm.org/citation.cfm?id=3341907

[26] Ayushi Rastogi and Nachiappan Nagappan. 2016. Forking and the Sustainability of the Developer Community Participation–An Empirical Investigation on Outcomes and Reasons. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE, 102–111.

[27] Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. 2013. Software clone detection: A systematic review. *Information and Software Technology* 55, 7 (2013), 1165–1199.

[28] Gregorio Robles and Jesús M. González-Barahona. 2012. A Comprehensive Study of Software Forks: Dates, Reasons and Outcomes, See [14], 1–14. https://doi.org/10.1007/978-3-642-33442-9_1

[29] Guillaume Rousseau, Roberto Di Cosmo, and Stefano Zacchiroli. 2019. *Growth and Duplication of Public Source Code over Time: Provenance Tracking at Scale*. Technical Report. Inria. https://hal.archives-ouvertes.fr/hal-02158292.

[30] Chanchal Kumar Roy and James R Cordy. 2007. *A survey on software clone detection research*. Technical Report 115. Queen's School of Computing. 64–68 pages.

[31] Diomidis Spinellis. 2005. Version control systems. *IEEE Software* 22, 5 (2005), 108–109.

[32] Stefan Stanciulescu, Sandro Schulze, and Andrzej Wasowski. 2015. Forked and integrated variants in an open-source firmware project. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 151–160.

[33] Ferdian Thung, Tegawende F Bissyande, David Lo, and Lingxiao Jiang. 2013. Network structure of social coding in GitHub. In *2013 17th European Conference on Software Maintenance and Reengineering*. IEEE, 323–326.

[34] Jason Tsay, Laura Dabbish, and James Herbsleb. 2014. Influence of social and technical factors for evaluating contribution in GitHub. In *Proceedings of the 36th international conference on Software engineering*. ACM, 356–366.

[35] Yue Yu, Huaimin Wang, Vladimir Filkov, Premkumar Devanbu, and Bogdan Vasilescu. 2015. Wait for it: determinants of pull request evaluation latency on GitHub. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 367–371.

[36] Shurui Zhou, Bogdan Vasilescu, and Christian Kästner. 2019. What the fork: a study of inefficient and efficient forking practices in social coding. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 350–361.