

# Altered Histories in Version Control System Repositories: Evidence from the Trenches

Solal Rapaport\*, Laurent Pautet†, Samuel Tardieu‡, Stefano Zacchiroli§

LTCI, Télécom Paris, Institut Polytechnique de Paris, Palaiseau, France

Email: \*solal.rapaport@telecom-paris.fr, †laurent.pautet@telecom-paris.fr, ‡samuel.tardieu@telecom-paris.fr,

§stefano.zacchiroli@telecom-paris.fr

**Abstract**—Version Control Systems (VCS) like Git allow developers to locally rewrite recorded history, e.g., to reorder and suppress commits or specific data in them. These alterations have legitimate use cases, but become problematic when performed on public branches that have downstream users: they break push/pull workflows, challenge the integrity and reproducibility of repositories, and create opportunities for supply chain attackers to sneak into them nefarious changes.

We conduct the first large-scale investigation of Git history alterations in public code repositories. We analyze 111 M (millions) repositories archived by Software Heritage, which preserves VCS histories even across alterations. We find history alterations in 1.22 M repositories, for a total of 8.7 M rewritten histories. We categorize changes by where they happen (which repositories, which branches) and what is changed in them (files or commit metadata).

Conducting two targeted case studies we show that altered histories recurrently change licenses retroactively, or are used to remove “secrets” (e.g., private keys) committed by mistake. As these behaviors correspond to bad practices—in terms of project governance or security management, respectively—that software recipients might want to avoid, we introduce GITHISTORIAN, an automated tool, that developers can use to spot and describe history alterations in public Git repositories.

## I. INTRODUCTION

Distributed version control systems (DVCS), like Git [8], [24], are among the most popular tools used by software developers [35]. In a typical Git workflow, each developer works on a local project repository that contains the full development history and integrate code changes there, by adding new commits. Then, to share changes with collaborators, developers push new commits to public repositories and pull from there new commits done by teammates into their local repository. These back and forth commit exchanges continue as long as the project is active. The consolidated view of the project is available at any time from a canonical public repository, which is used by users and other downstream recipients to obtain the software source code.

For the most part Git repositories are append-only: developers add *new* commits to their repositories. But it is technically possible to *alter the previously recorded history*

of a Git repository [8, Chapter 7.6 Git Tools — Rewriting History] and modify old commits. Two common reasons for altering Git histories are: (1) “amending” a recent commit to change its content or metadata, and (2) “rebasing” a portion of VCS history to merge commits, split them, or avoid merges by “linearizing” development branches. Any change to a previously recorded commit results in one or more commits changing their public identifiers (cf. Section II for details).

History rewrites are useful to polish local and/or work-in-progress VCS branches before sharing it with others, and are even the default behavior in some development workflows [32]. For example, pull request (PR) branches are often rewritten in order to keep only a single commit in them that can be merged into the target branch when the PR is ready.

History alterations are however problematic when performed on public branches used by downstream recipients, for various reasons. First, they break the pull/push workflow [17], [26], which relies on the stability of commit identifiers as checkpoints for history exchanges. Second, by breaking commit identities, alterations create opportunities for supply chain attacks on repositories [21], [27]: together with a legitimate history rewrite, attackers can sneak in malicious changes, which will be difficult to audit due to the confusion caused by commit identity modifications.

## A. Contributions

Little is currently known about the **amount, characteristics, and impact of VCS history alterations in public code**. This is partly because it is an inherently difficult phenomenon to study empirically at scale: after rewrite, the VCS history *prior* to the alteration no longer exists and cannot be compared with what remains post-alteration. This work contributes to bridge this gap, by conducting the first large-scale quantitative and qualitative analysis of history alterations in public VCS repositories. Specifically, we address the research questions detailed below.

**RQ 1.** *How often are the histories of publicly available VCS repositories altered in an observable way?*

To solve the methodological problem of having access to VCS histories before and after alterations, we mine Software Heritage (SWH) [1], [11], the largest public archive of software source code, which maintains full copies of public code repositories, even across history rewrites.

This work was supported by France Agence Nationale de la Recherche (ANR), program France 2030, reference ANR-22-PTCC-0001 and partially supported by Agence de l’Innovation de Défense – AID – via Centre Interdisciplinaire d’Etudes pour la Défense et la Sécurité – CIEDS – (project 2021 – FARO). It was made possible by Software Heritage, the universal source code archive: <https://www.softwareheritage.org>.

We analyze 111M (millions) public Git repositories archived from major development forges like GitHub as well as many GitLab public instances. We identify 12.5B (billions) altered histories, detected by the observable that at least one commit reachable from a previously archived repository state is missing from a later one. The phenomenon is quantitatively non-negligible: it impacts 1.22M repositories.

**RQ 2.** What (a) branches and (b) repositories are impacted the most by history alterations?

We breakdown the quantitative findings from RQ 1 by branches, to understand how the phenomenon relates to development workflows; and by repository popularity, to understand if underused and potentially lower-quality repositories are more impacted or not.

We then turn our attention to qualitative aspects:

**RQ 3.** When the history of a VCS repository is altered, which commit parts undergo modifications?

We categorize what is altered using a novel taxonomy that captures whether files (i.e., versioned files) or metadata (e.g., timestamps, commit messages, etc.) are changed. We find that 13.3% history alterations concern metadata-only changes, in most cases impacting multiple metadata at once, whereas 76.8% alterations include changes to files and/or directories.

**RQ 4.** Are there recurrent patterns of file alterations among observed VCS history rewrites?

To evaluate concrete risks for software developers, we look into the raw results obtained from RQ 3 for recurring patterns of files that are frequently part of history alterations.

We analyze two specific scenarios: (a) removal of “secrets”, like private keys, and (b) license changes occurring in appropriately-named files (e.g., LICENSE). Instances of (a) denote subpar security diligence on the part of the developers; (b) instances are also problematic because, while license changes can legitimately happen during the lifetime of a project, they should not happen *retroactively* by altering VCS history, as that might results in users who lack old copies of the repository losing rights. To help developers spotting these and other problematic history alteration patterns, we explore one final question:

**RQ 5.** Can we design and implement an automated tool to audit and detect history alterations in public repositories?

We demonstrate how our tool GITHISTORIAN performs both efficient and accurate analysis on very large-scale archival datasets. It enables developers to check if repositories of their interest underwent history alterations, providing detailed information about (and optional filtering on) what was changed, when the alterations occurred, and which commits were affected. This enables project maintainers to quickly detect suspicious past alterations and investigate further. GITHISTORIAN can also be integrated into CI/CD pipelines to provide automated alerts upon detection of history alterations.

## B. Data availability statement

A full replication package containing the data and code for the experiments presented in this paper, as well as the

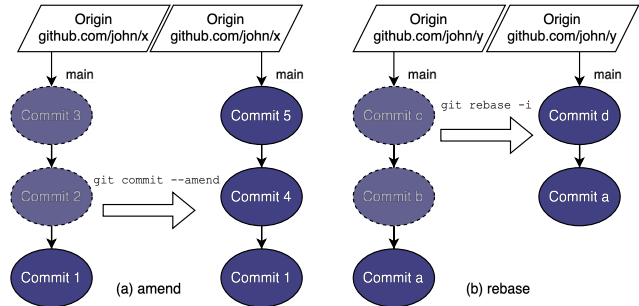


Fig. 1. Common scenarios leading to history alterations in Git: (a) commit amendment, (b) interactive “rebase”. Grayed out commits are no longer reachable after alteration. Arrows point from newer to older commits.

GITHISTORIAN tool, is available from Zenodo and Software Heritage [31].

## II. BACKGROUND

*a) Git data model:* The data model of Git, like most other DVCS [7], is a Merkle [25] direct acyclic graph (DAG), where nodes are used to represent commits and other types of source code artifacts. Each DAG node is identified by a cryptographic checksum computed recursively on its content and metadata which, for commit nodes, include the identifiers of previous commits. The full VCS history of a project hence forms a connected graph whose integrity can be verified efficiently by only considering the identifiers of outer “root” nodes: if they match a previous known state, then all previous commits reachable from them have not been altered since.

*b) History alteration with Git:* Figure 1 shows two common use cases of history rewrites in Git: (a) amending and (b) rebasing. The “amend” functionality allows developers to retroactively modify parts of commits that were already recorded in the version history: *commit metadata* (e.g., timestamp, message, author name or email, etc.) and/or *commit content* (e.g., adding or removing files). In Figure 1(a), the user modifies commit 2, producing the new commit 4. After history alteration, due to how Merkle identifiers work, commit 2 disappears from the graph while commit 4 is added to it. In the example, since commit 3 is based on commit 2, commit 3 is *also* modified and becomes commit 5. History alteration can therefore induce a “snowball effect”, where all the commits that transitively reference modified ones get new identities as well. If commit 2 had one million commits (transitively) referencing it instead of one, they would *all* disappear from the repository and be replaced by new ones.

Figure 1(b) shows the other common use case of Git history rewrites: “rebasing”. In this example, commits c and b are merged into a single commit: they both disappear and are replaced by new commit d. In other use cases, rebase can also split commits or attach them to different previous commits.

*c) Terminology:* A *repository snapshot* (or simply *state*) is the observable state of a repository at a given point in time. Intuitively, it corresponds to the set of all commits in it.

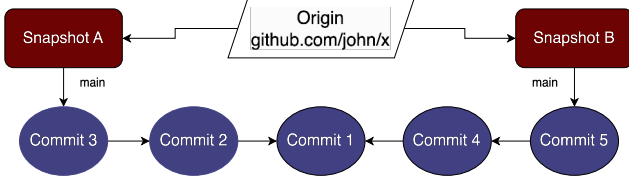


Fig. 2. Repository state before and after `git commit --amend` of Figure 1(a). Both histories are preserved by Software Heritage and share common commits.

We say that a repository  $R$  underwent a *history alteration* if there exist two subsequent repository snapshots  $S_1, S_2$  of  $R$  such that  $\exists c_i, c_i \in S_1 \wedge c_i \notin S_2$ , i.e., at least one commit is present in the first snapshot but missing from the second. Multiple history alterations can be observed for the same repository if multiple snapshot pairs  $S_i, S_{i+1}$  denote history alterations.

An *altered commit* is a commit impacted by a history alteration, as detected by the fact that it is missing from a subsequent repository snapshot, but present in a previous one. Due to the snowball effect, potentially many commits will be altered as a consequence of a single history alteration, but a minimum of one commit must be (otherwise the alteration will not be detected).

For each history alteration, we identify one or more *root cause commits* as the oldest commits, in topological order, among all altered commits. Root cause commits are commits that have been *directly* altered: their identity changes are not a mere consequence of the modification of previous commits.

d) *Software Heritage*: (SWH) [1], [11] archives the development history of more than 370 million projects hosted on major development forges like GitHub and GitLab.<sup>1</sup> Each archived project is identified by its *origin*: the repository URL. SWH crawlers periodically visit each origin, taking complete snapshots of the repository state and storing collected artifacts in a global Merkle DAG. As shown in Figure 2, if commits disappear, e.g., due to history alterations, from a repository between two archival visit, they will still be reachable from the *previous* snapshot of the same origin. We rely on this feature as the basis for our empirical experiments, whose methodology we describe next.

### III. METHODOLOGY

To answer the stated research questions, we followed an empirical methodology comprised of 4 phases:

- 1) we collect the largest existing corpus of periodically crawled public repositories for analysis (addressing RQ 1);
- 2) second, we devise an experimental protocol to detect history alterations (addressing RQ 1 and RQ 2);
- 3) we create a taxonomy to categorize history alterations and understand the reasons behind them (RQ 3);
- 4) based on the observed recurrent patterns of file name alterations, we illustrate the benefits of our approach and

<sup>1</sup><https://archive.softwareheritage.org>, accessed 2025-05-23.

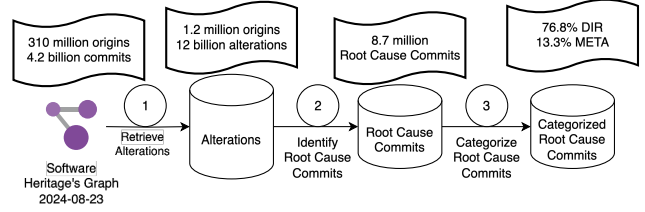


Fig. 3. Methodology for detecting and categorizing history alterations.

dig further into two case studies: history alterations to remove “secrets” committed by mistake, and alterations impacting license files (RQ 4).

#### A. Data collection and repository selection

In order to analyze a large amount of public software repositories, we start from the Software Heritage (SWH) graph dataset [30]. We retrieved the most recent dataset version available at the time of our experiments: timestamp 2024-08-23.<sup>2</sup> It contains the VCS history of more than 310 M (millions) repositories and packages (“origins” in SWH terminology). We will detect history alterations by comparing successive snapshots of the same origin, as detailed below in Section III-B. To avoid under-estimating the ratio of repositories that witness history alterations, we exclude those visited by SWH crawlers less than 2 times (199 M). We also exclude origins that are not Git repositories (0.019 M), due to the popularity of Git and in order to better relate findings to practices. This leaves us with 111 M origins where we can potentially find evidence of history alterations.

To answer RQ 2(b), we will group repository by popularity and use rely on GitHub “stars” as popularity indicator [6]. Although the number of stars can be artificially inflated by malicious actors [15], it is still a relevant metric in our case: a large number of stars increases repository visibility, making it more worthy of scrutiny, including to detect history alterations. We rely on GitHub star counts made available by SWH, that also crawls platform-specific metadata like these. Due to the uneven popularity of development platforms, in the following we restrict analyses that depend on popularity to GitHub repositories; other analyses are conducted on all repositories archived by SWH, independently of the platform.

#### B. General framework

Figure 3 shows the experimental approach we devised to detect and categorize history alterations. For each SWH origin with at least two visits, and for each Git branch in them, we compare all pairs of successive snapshots to identify commits from the earlier snapshot which are missing from the later one. In this set of commits, we identify the *root cause commits* according to the definition given in Section II. We implement this step in practice with custom code that mine locally the compressed graph representation of the SWH archive [5], via its Rust API. This detection framework enables us to address

<sup>2</sup><https://docs.softwareheritage.org/devel/swh-dataset/graph/dataset.html>

TABLE I  
BRANCH NAMES UNIFICATION

Branch names or patterns	Unified name
main, master	main
dev, devel, develop, development	development
pull/number/head	pull request
renovate/anything	renovate

TABLE II  
ROOT CAUSE COMMITS BY CATEGORY

Category		#Commits	Total
META	Author	667 424	1 160 725
	Message	691 060	
	Date	769 287	
	Committer	649 937	
	Committer Date	864 607	
	Other	55 638	
DIR	File Modified	5 469 080	6 693 247
	File Removed	1 287 304	
	Content Split	763 369	
Different Branch Name		866 113	866 113

both RQ 1 (by counting alterations) and RQ 2 (by analyzing branch-level patterns).

In Git, different branch naming schemes are used to represent the same concept. For example, branches `main` and `master` are both used to designate the main stable branch of a repository, while branches hosting GitHub pull requests start with `pull/`. A popular bot used to maintain project dependencies up-to-date uses the `renovate/` prefix, and development branches often start with `dev`. We unify together branches used for the same purpose, to provide aggregate statistics in the following (addressing RQ 2 (a)). Details about how we unified branches together are shown in Table I.

### C. History alteration detection and categorization

We categorize the root cause commits of history alterations using the custom taxonomy shown in Table II (we found no preexisting taxonomy for this in the literature), which allows to capture alterations to both metadata and file (and/or directory) content. This taxonomy directly addresses RQ 3 by systematically characterizing what commit parts undergo modifications. The taxonomy was developed via an iterative mixed method: sampling uncategorized history alterations, manually comparing their before/after states, capturing the individual data that differed (leading to introducing additional categories if needed), applying all categories to the entire dataset, and repeating until all data was categorized. Starting from the simplest case: a (root cause) commit is categorized as *Different Branch Name* if its content and metadata have *not* been altered, but it is now found only in a different branch than the previous one.

A commit is categorized as *META* if it has been replaced by a new commit with strictly identical files and directories *content*, but different metadata. This categorization is further refined to indicate which metadata fields have been altered. Multiple metadata fields can be modified in a single history alteration.

A commit is categorized as *DIR* if the commit file/directory content was altered or deleted, in part or as a whole. In this case, we do not analyze metadata changes, as there is no single reference commit to compare them with. The *DIR* categorization is further refined to describe what happened to altered content:

- *Content Split*: the content of each file that existed in the altered commit is still in the repository, but is now split into several other commits. This is exclusive of other subcategories in “*DIR*”.
- *File Modified*: at least one file has been modified.
- *File Removed*: at least one file was removed, and was not found in other commits.

To distinguish among these we look in the newest snapshot at commits starting from the children of the parents of the root cause commit, and proceed for 10 generations (commit tree depth). If, after 10 generations of descendants, or if there are no more descendants, one of the file paths cannot be found, the file is considered removed and the commit is marked as such. We chose 10 generations heuristically, as a trade-off between analysis time and diminishing likelihood that a file will reappear at the same path without being a distinct file.

### D. Case study design

With the taxonomy developed thus far we are able to peek into recurring patterns of history alterations at various levels, including which files are modified retroactively in the *DIR* category. Based on this, we looked more in-depth into two recurring patterns of VCS history alteration, addressing RQ 4.

a) *Secrets suppression*: The first case study focuses on secrets removed (category: File Removed) from a repository by altering its history. With “secret” we mean potentially sensitive information that should never have been shared via a public repository. We detect this based on popular file names used for storing private keys of passwords, like `id_rsa` (without `.pub` extension), `secring.gpg`, etc. The full list is available from the replication package [31]; further examples in Section IV-2a.

b) *Retroactive license changes*: The second case study identifies retroactively modified license files on the main branch (after branch unification) of a repository. It is considered bad practice to *retroactively* alter the license of an open source project, as it makes impossible for novel users to obtain the software under its previous license, possibly to fork it.

To detect license changes, we look at altered files (category: File Modified) containing `LICENSE` (or `LICENCE`, ignoring case) in their name. We then run ScanCode [28], [33], a state-of-the-art license detection tool, on the files before and after alteration, considering only the highest-confidence result and only if it is above 90%. We obtain this way two *sets* of licenses (before/after) that we further categorize as follows to evaluate the magnitude of the change:

- *License Update*: the set of unversioned licenses (e.g., GPL, Apache) remain the same, whereas that of versioned licenses changes (e.g., GPL-2 → GPL-3). Note that, even

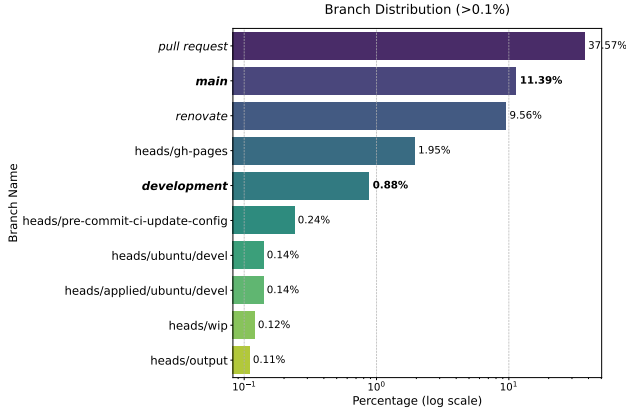


Fig. 4. Most impacted branches by Altered Histories (> 0.1%)

in this lower-impact case, there is no valid reason to make the change *retroactively* by rewriting Git history.

- *Partial Change*: some (unversioned) licenses have changed, e.g., GPL has been added, MIT removed.
- *Full Change*: all (unversioned) licenses have changed.

#### IV. RESULTS

##### RQ 1: how often are VCS histories altered?

Across our entire corpus, we observed 12 542 848 352 alterations spread across all analyzed branches. In terms of repository count, 1218 547 contained at least one altered commit, representing approximately 1.10% of all examined repositories. While the proportion is modest in relative terms, it is not negligible in absolute ones: users of more than 1 M repositories can—and will in the future assuming future stability—use public Git repositories that underwent history alterations during their lifetime.

At the same time, the low percentage indicates that history rewriting remains an exception rather than standard practice, suggesting that common Git usage practices adhere to the expectation of immutable version control histories.

##### RQ 2a: what branches are altered the most?

We now look into where history alterations happen, starting within repositories at the granularity of specific branches. For this analysis we consider branch names after the unification of Table I.

Figure 4 presents the distribution of history alterations by branch, showing the top-10 branches. The results reveal some patterns that align with popular development expectations, but also highlight more concerning cases. Indeed, as one might expect, pull request branches represent the most frequent target of history alterations, with 37.57% of all observed alterations. This pattern reflects the common practices where contributors refine their pull requests through interactive rebasing, commit squashing, etc. to produce a clean history before integration.

Other high-ranked branches give cause for concern. *Main* branches (after unification, hence spanning both *main* and *master*) are in general expected to be long-lived and stable,

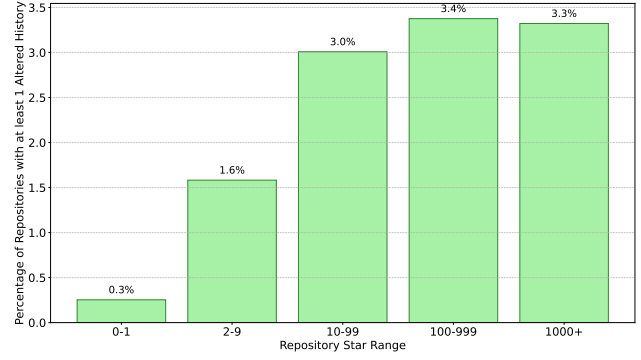


Fig. 5. Proportion of Repositories with Altered Histories per Star Range

but still experienced history alterations in 11.39% of the cases we observed, ranking as the 2nd-most impacted category. Additionally, development branches account for 0.88% of observed alterations, ranking 5th overall. Depending on the project and workflow, development branches might be expected to be stable or not.

The 3rd-most impacted category at 9.56%, *renovate* branches, represents automated dependency update workflows. These alterations stem from bot behavior that rewrites commits when dependency proposals evolve prior to merging, making history modification an inherent characteristic of automated maintenance processes.

Even though, around 50% of history alterations follow a known practice, these findings raise significant implications for repository integrity and developer trust. History alterations on main branches compromise the fundamental assumptions of version control systems, potentially breaking downstream dependencies, invalidating signed commits, and disrupting reproducible builds [10]. The 12% alteration rate for main branches suggests that a substantial subset of repositories may pose reliability risks for users who depend on stable commit references.

##### RQ 2b: what repositories are altered the most?

Not all repositories are born equal: repositories created by student just because their teacher said so do not have as many downstream users as *torvalds/Linux.git*. One might then wonder if repository popularity correlates with the amount of history rewrites, under the hypothesis that maturity leads to a more limited use of disruptive VCS practices. To explore this we join the set of repositories that experienced at least one alteration with their amount of GitHub stars. From 1218 547 altered repositories, we remove 283 203 origins with unknown amount of star. We observe the following star distribution:

- 0 to 1 star: 676 252 origins (72.3%)
- 2 to 9 stars: 115 431 origins (12.3%)
- 10 to 99 stars: 82 519 origins (8.8%)
- 100 to 999 stars: 43 416 origins (4.6%)
- 1000 and plus stars: 17 726 origins (1.9%)



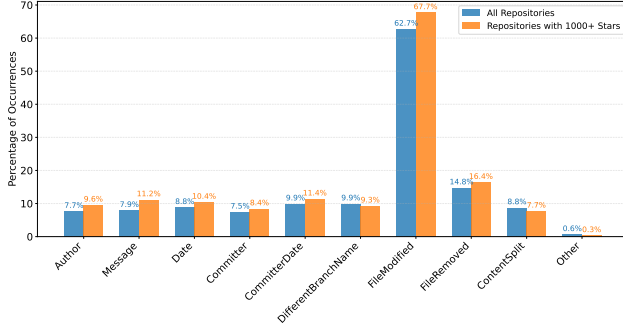


Fig. 6. Distribution of Categories: All vs. Popular Repositories

Figure 5 shows the percentage of origins with detected history alteration for each bucket. The analysis reveals a pronounced relationship between repository popularity and the likelihood of history alterations. Repositories with minimal popularity (0 to 1 star) exhibit the lowest alteration rate at 0.3%. This rate increases to 1.6% for repositories with 2 to 9 stars, 3.0% for those with 10 to 99 stars, and then stabilizes at approximately 3.4% for repositories with 100 to 999, 3.3% for those with 1000 and plus stars.

It is hence *not* the case that less popular repositories experience more rewrites than more popular ones. Rather, as repositories gain visibility, the likelihood of history modifications increases substantially. Even very mature and popular repositories (100+ stars) experience a non-negligible amount of history alterations, which is potentially concerning in terms of repository auditability.

We also looked into which branches are altered the most and compared results with repository popularity. In very popular repositories (100+ stars) history alterations happen for the most part in pull request branches (56%) but remains non negligible in *main* branches (1.8%). This finding has significant implications for dependency management, reproducible builds, and the general reliability assumptions that developers make when incorporating external repositories into their workflows, irrespectively of their popularity.

#### RQ 3: which parts of commits are altered?

We apply the taxonomy of Table II looking first for metadata-only changes (category: META), then for file/directory changes (category: DIR). The results for each category are displayed in Figure 6, which shows the proportion of altered histories in each category—META categories on the left, DIR categories on the right—for both the entire corpus and for the subset of most popular GitHub repositories (1000+ stars).

1) *Metadata attributes changes (META)*: Among the 8 720 085 identified root cause commits, 1 160 725 instances (13.3%) exhibited modifications only to commit metadata, leaving underlying files and directories unaltered.

Digging into specific metadata categories shows that committer date represents the most frequently modified attribute across both the complete dataset and the subset of repositories

TABLE III  
TOP 20 MOST ALTERED FILE NAMES

Rank	File Name	Count	Rank	File Name	Count
1	default.nix	13 401 935	11	CHANGELOG.md	5 001 971
2	package.json	12 594 717	12	__init__.py	4 859 517
3	Makefile	10 615 195	13	metadata.xml	4 477 947
4	README.md	9 969 599	14	CMakeLists.txt	4 380 963
5	index.js	9 129 894	15	package.py	4 336 900
6	Manifest	8 891 876	16	APKBUILD	3 375 064
7	pom.xml	7 080 355	17	template	3 122 049
8	index.html	6 297 643	18	BUILD	3 018 131
9	meta.yaml	5 425 074	19	index.ts	2 764 520
10	index.md	5 372 929	20	LICENSE	2 762 147

with 1000 or more stars. We do not observe significant variations in the modification patterns of the two sets: metadata modification practices remain uniform regardless of popularity.

We also conducted a more in-depth analysis of the co-occurrences of metadata changes, i.e., META cases that involve changes to 2 or 3 metadata fields at the same time. The most frequent alteration for 2 fields at once is the change of author date, which always coexist with a change to the committer date—not doing so would indeed be surprising.

The most frequent triplet of metadata fields changed together in the global dataset that occurs 85% of the time is when the commit message changes; in most of those cases author and committer dates change as well. This result corresponds to the expected behavior when modifying a commit message.

2) *Directory content changes (DIR)*: We categorized 6 693 247 altered commits based on changes to the content of the files or directories. The dominance of the category *File Modified* is significant in both the global dataset and the subset of popular repositories that display a proportion higher than 60%. In both datasets the occurrence of files being modified in a history alteration is more than 4 times higher than the occurrence of files being removed. Developers who alter a file in the history are 4 times more likely to modify rather than deleting, renaming or moving it around.

The final subcategory corresponds to commits moved between branches: *Different Branch Name*, with 866 113 commits. The most plausible explanation for this are merges of so-called “feature branches”, which are later removed from the repository; this behavior is still problematic for direct users of feature branches, but are understandable if those branches are explicitly documented as volatile.

#### RQ 4: are there recurrent file patterns in alterations?

As we have seen while answering at RQ 3, file alterations are the most common forms of VCS history alteration. This begs the question of *which* files are being modified or removed. To answer RQ 4 we extract the file name part of all file paths involved in history alterations of category DIR. Table III shows the top-20 of most commonly altered file names.

The most frequently altered files are predominantly configuration and build-related artifacts. *default.nix* (#1), *package.json* (#2), *Makefile* (#3), *pom.xml* (#7), *CMakeLists.txt* (#14), and *APKBUILD* (#16) represent different build systems and package managers. This suggests

that dependency updates, version bumps, and build configuration changes are primary motivations for altering the history.

The high frequency of files like `default.nix`, `package.json`, and `meta.yaml` matches the workflows of automated dependency management tools (like Renovate, Dependabot [16], or Nix update bots) that systematically modify these files and subsequently rewrite commit histories when adjustments are needed.

Files like `README.md` (#4), `CHANGELOG.md` (#11), and `index.md` (#10) represent project documentation, indicating that documentation improvements are another common reason for history rewrites. The presence of entry points across different ecosystems (`index.js` #5, `index.html` #8, `index.ts` #19, `__init__.py` #12) suggests that main application files are frequently altered, reflecting refactoring or structural changes that seem to require frequent history cleanups.

Lastly, the appearance of `LICENSE` (#20) and `metadata.xml` (#13) indicates that legal compliance and project metadata corrections constitute a notable category of history alterations, reflecting retroactive license changes or potential metadata correction efforts.

This pattern analysis reveals that history alterations predominantly target infrastructure, configuration, and metadata files rather than core application logic, suggesting that maintenance activities and automated tooling are primary contributors to the observed history alterations.

*a) Case study: removing “secrets” from history:* Across all repositories, we identified 13M history alterations categorized as *File Removed*, involving files whose names commonly denote private information that should generally not be distributed publicly, like private keys, private certificates, and password. These alterations occur across 75 k (thousands) different repositories in our dataset.

Generic naming patterns demonstrate the highest frequency of occurrence, with files containing “key” and “secret” accounting for 6.9M and 805 k altered files, respectively. Manual examination of a representative random sample confirmed that removed files contained sensible private information, in files such as `secret.yaml` (containing authentication credentials) and `samlKey.jks` (storing keys for the deployment of production environments).

SSH private keys were less popular, probably due to the fact that platforms like GitHub can nowadays block Git pushes containing them, but are still not absent from our dataset: we identified 108 removed files containing RSA private keys, spanning 23 origins.

These findings demonstrate that despite developers’ attempts to remediate secret exposure through history alteration, such sensitive information remains recoverable through history digging in archives. When developers inadvertently publish private information, removing it from *their* repository history represents only partial remediation. Complete security restoration requires regenerating or rotating all exposed tokens, passwords, and cryptographic keys. But if they do so, there

is arguably little point in rewriting the repository history afterwards.

It is important to remind here that via our methodology we can only detect alterations occurring between captured repository snapshots. Consequently, our results correspond to either cases where developers were unfortunate enough to have snapshots captured immediately following secret commits, or instances where significant time elapsed between exposure and remediation attempts.

*b) Case study: retroactive license changes:* In our second case study we focus on retroactive license modifications, where developers alter VCS history changing the content of license-denoting filenames (e.g., `LICENSE`). These changes are quite frequent in our experiments, as evidenced by the presence of `LICENSE` among the top-20 most frequently altered file names in Table III.

We identified 796 972 altered license files across the main branches of our dataset, across 32 169 different repositories, with 76 of them having 1000+ stars on GitHub. Among these, 719 196 instances involved file renaming, relocation, or deletion, while 77 776 cases represented content modifications.

Using ScanCode as described in Section III-D, we successfully identified the set of before/after-alteration licenses for 65 688 history alterations involving license files. Our analysis reveals that 79% of alterations are relatively minor license updates, implicating at most changes in license versions. Further manual analysis indicates that copyright modifications (e.g., in the name of the author) are a common occurrence in this category. 14% of alterations represent full license changes, encompassing both transitions from more permissive (e.g., MIT) to more restrictive (e.g., GPL) licenses and vice-versa. Only 5% of alterations constitute partial changes, which we attribute to the limited number of repositories (8024 alterations) detected with multiple license families, thereby reducing the representativeness of partial change scenarios.

These findings are concerning for downstream users of open source products, because while *legally* most open source software licenses are irrevocable and meant to grant specific rights to users forever, benefiting of those rights require practical access to a version of the software released under a given license. Retroactively changing version history can constitute an attempt to make old software versions, under an old license, disappear from public circulation, inhibiting previously available rights. This is even more concerning for industrial users, who tend to appreciate, and expect, legal stability in the open source software they depend on.

## V. DETECTING AND AVOIDING ALTERED HISTORIES

VCS history alterations go unnoticed by everyone, except for downstream users who happen to have retrieved and kept a repository version before the alteration, and also `git pull` after the alteration, with branch tracking in place for the affected branches. Few users satisfy these conditions. For example, it is not uncommon for CI/CD pipelines that depend on external software to perform a fresh `git clone` of repositories of interest, making it impossible to detect

history alterations. Nonetheless, some history rewrite patterns might be causes of concerns for downstream users of affected repositories: they might hint at bad maintenance practices or, worse, be evidence of active tampering by malicious actors. To help both downstream users and upstream maintainers worried about history alterations, we answer RQ 5 practically, by designing, implementing, and showcasing GITHISTORIAN: a practical tool capable to detect, describe, and avoid (if desired) repositories that witnessed VCS history alteration.

#### A. Design

The design of GITHISTORIAN satisfies three core requirements: scalability for analyzing thousands of repositories, accuracy in detecting various types of history alterations, and usability for both interactive analysis and automated monitoring. The system architecture consists of four main components: (1) a PostgreSQL database storing preprocessed history alteration data derived from Software Heritage snapshots,<sup>3</sup> (2) a command-line interface (CLI) for interactive repository analysis, (3) a caching mechanism to optimize repeated queries, and (4) Git hook integration for continuous monitoring of repository changes.

The detection mechanism operates by querying the database for known alterations. The tool supports branch-specific analysis, allowing users to focus on main development branches or examine all branches comprehensively.

For automated monitoring, the tool employs Git hooks that trigger checks after repository updates (`post-merge`) and branch switches (`post-checkout`). This design ensures that developers are immediately notified when working with repositories that have experienced history alteration, without requiring manual intervention.

#### B. Implementation and CLI

GITHISTORIAN is implemented in Rust, using the `sqlx` crate for type-safe database interactions and `tokio` for asynchronous operations, enabling efficient handling of database queries even with large datasets like ours. The implementation is available as part of the replication package of this paper [31].

The main functionalities can be accessed via the main CLI commands:

**Database Management:** The `load` command handles the initial dataset ingestion into the local database, with parallel processing support. This allows the efficient loading of datasets containing billions of alteration records.

**Repository Analysis:** The `check` command requests auditing a specific repository to detect history alterations. It queries the local database as needed, and formats results for human consumption. It is possible to analyze main branches only, development branches only, or all branches.

**Automated Monitoring:** The `attach` command installs Git hooks that call the `check-cached` command. This cached

<sup>3</sup>The local database is required because SWH currently does not provide a remote API answering the queries needed to detect history alterations. This requirement can be lightened in the future by either SWH operators adding this service, or by some third parties providing it, consuming the datasets that SWH publish periodically, as we did one-off for our experiments.

variant implements smart result caching based on repository URL, target branch, and dataset version, avoiding redundant database queries whenever possible.

**Caching Strategy:** Results are cached by repository URLs, with cache validity determined by dataset version comparison. This approach ensures that results remain accurate when new alteration data becomes available while providing immediate responses for repeated queries if desired, or no response to avoid redundancy.

Hook scripts include contextual information about the triggering event (pull, merge, or checkout) and provide clear user feedback about the analysis results.

#### C. Example

Consider a developer working on a project who wants to ensure their repository dependencies did not undergo history alteration. After installing and loading data into GITHISTORIAN, one can verify the absence of alterations in any repository of interest as follows:

```
$ git-historian check https://github.com/example/project
--branch main --verbose
Connected to the database!
Found 2 altered history records for
'https://github.com/example/project'

Altered History Records:
Record #1:
  Branch Name: refs/heads/master
  Altered Commit: swb:1:rev:alb2c3d4e5f6789...
  Snapshot Destination: swb:1:snp:abcd1234...
  Sub Category: FileModified

Record #2:
  Branch Name: refs/heads/dev
  Altered Commit: swb:1:rev:f6e5d4c3b2a1098...
  Snapshot Destination: swb:1:snp:efgh5678...
  Sub Category: CommitterDate

File Modifications:
Record #1:
  Branch Name: refs/heads/master
  Altered Commit: swb:1:rev:alb2c3d4e5f6789...
  File Path: src/security/auth.py
  Status: Modified

Results saved to: ~/.local/state/git-historian/
altered_history_example_project_20241201_143022.txt
```

The repository underwent two types of history alteration: a file modification and a metadata alteration to change the commit date. The verbose output shows that a security-related file was modified in one of the altered commits, information that could be crucial for security auditing.

For automatic monitoring in the future upon clone or pull actions, the developer can attach the tool to their local repository (or CI stateful repositories) like this:

```
$ cd /path/to/local/repo
$ git-historian attach . --branch main --verbose
git-historian successfully attached to repository
Repository: .
Remote URL: https://github.com/example/project
Hooks installed:
- post-merge (triggered after git pull)
- post-checkout (triggered after git checkout)
```

## VI. DISCUSSION

### A. Implications for software development practices

Our findings challenge the conventional wisdom about version control best practices in several ways. The paradoxical correlation between repository popularity and alteration frequency indicates that current development workflows may



be fundamentally at odds with the immutability principle underlying version control systems.

While alterations on pull request and dependency management branches follow accepted practices, their normalization may be creating cultural acceptance that inadvertently extends to main branches. This gradient of acceptability poses risks for repository integrity standards across the development lifecycle.

The substantial presence of main branch alterations represents a systemic challenge to fundamental software engineering assumptions. Beyond the immediate technical impacts on CI/CD systems and semantic versioning, these practices undermine the cryptographic integrity guarantees that many security and compliance frameworks depend upon. Organizations relying on commit signatures or audit trails may unknowingly operate with compromised assurance levels.

#### *B. Security and compliance implications*

Our secret removal analysis reveals deeper organizational security governance issues beyond immediate credential exposure. The prevalence of post-hoc secret removal suggests systematic failures in preventive security measures, pre-commit hooks, developer training, and secure development workflows. Organizations discovering secrets in their histories face a false choice between public exposure and historical integrity.

More concerning is the implicit assumption that history cleaning provides adequate remediation. Our detection capabilities demonstrate that “deleted” secrets remain discoverable through historical analysis, yet many organizations may believe their incident response is complete after history rewriting.

The licensing case study exposes novel legal risks in open source governance. Retroactive license changes through history alteration create ambiguous legal precedents where different project versions operate under potentially conflicting terms. For organizations with complex supply chain dependencies, these alterations introduce uncertainty about intellectual property rights and compliance obligations that traditional license scanning cannot detect.

#### *C. Tool and ecosystem implications*

The automation-driven nature of many history alterations reveals an emerging disconnect between tool capabilities and user understanding. Modern dependency management systems normalize history rewriting as an implementation detail, potentially creating user expectations that extend beyond appropriate use cases. This normalization effect may contribute to the casual application of history alteration techniques in contexts where they are inappropriate.

Security tooling faces challenges in distinguishing between legitimate maintenance and potentially suspicious activity, requiring sophisticated context analysis beyond simple pattern matching.

The research ecosystem faces a methodological challenge as our findings demonstrate that significant portions of repository-based studies may analyze sanitized rather than authentic development data. This has implications not only for historical accuracy but also for the reproducibility of software

engineering research that relies on commit-based metrics and temporal analysis.

#### *D. Broader research and community implications*

These findings challenge several assumptions underlying current software engineering research and practice. The prevalence of modifications suggests that a non-trivial portion of repository-based research may be analyzing sanitized or restructured data rather than authentic development histories.

For the broader open-source ecosystem, our findings raise questions about trust, reproducibility, and the long-term integrity of collaborative development platforms. The development of standards, best practices, and tooling to address these challenges represents an important area for future community investment.

#### *E. Supply chain attack obfuscation implications*

History alterations present a concerning attack vector for software supply chain compromise. Attackers could inject malicious code while using history rewriting to erase evidence, making detection through conventional security auditing extremely difficult. The normalization of history rewriting practices we documented creates perfect cover for malicious actors to make backdoors appear as legitimate maintenance activities.

The scale of alterations we observed—affecting millions of repositories—suggests this represents a significant blind spot in current supply chain security practices. Traditional security measures focus on current repository states but may miss attacks exploiting the temporal dimension of version control history.

### VII. THREATS TO VALIDITY

#### *A. Construct validity*

History alteration is, by design, a destructive operation. As such, the only way to observe it faithfully without information losses, would be on developer machines conducting a user study, dramatically reducing the scale of the experiment. In this work we took the opposite approach of conducting a very large-scale experiment, relying on Software Heritage (SWH) to collect and preserve repository snapshots, which we then compare two-by-two to find traces of history alterations.

As such, we depend on the SWH archival frequency of public repositories: we can only observe history alterations that are visible between successive visits of a repository. Malicious or accidental short-lived alterations might be missing from our analysis, if they have been fixed by reverting to an unaltered history version in between two archival visits of the same repository. Also, we cannot distinguish *multiple* history alterations that happen between two visits of the same repository: we will recognize them as a single alteration, merging together what changed in all of them. To do better, one would need almost real-time archival of a large enough amount of public code repositories—something that, to the best of our knowledge, no dataset provides today—or, alternatively, an integrated insider study of a specific development platform (e.g., GitHub), which would introduce a selection bias.

Metadata changes are not detectable with our methodology when file changes *also* occur, due to the lack of a commit to compare against. As such, our analysis overlooks metadata changes that co-occurred with file ones. We could nonetheless characterize them for more than 1 M commits, but we make no claim about the rest.

When analyzing file changes we limit the search horizon to identify if and where previous content went, searching up to 10 commits deep. It is possible that content might be found with a larger horizon. However, it would become debatable whether a file reappearing “many” commits away from a change is to be considered the same file or a new one.

### B. External validity

We only considered Git repositories in this study and we do not claim further generality or applicability to other DVCS. SWH archives code maintained using other DVCS as well, so it is theoretically possible to use the same dataset of this work to analyze others, possible for comparison purposes. However, it would be difficult to disentangle the specificities of each DVCS, e.g., in terms of branch naming, for a proper comparison. On the other hand, some DVCS allows for non-destructive history updates, which keeps track of previous history states. Analyzing and comparing them with our findings would be interesting, even if by definition they would rule out some of the use cases considered in this work, like secret removal.

Finally, we observe that Git is a DVCS, but is also used as “storage backend” for other independent DVCS, like Jujutsu (j j) [18], where history alterations are used more liberally. We did not attempt to detect and separate “regular” Git repositories from these cases. This might impact our analysis, but we expect it to remain quantitatively marginal to this day.

## VIII. RELATED WORK

### A. History alterations: motivations and handling

The Pro Git Book [8] mentions history rewriting as a “great thing” and lists the actions one can perform *before* sharing development history with others: modifying files, changing commit messages, reordering, squashing, splitting, or removing commits. The ability to modify files is essential for removing erroneously committed sensitive information [3]. Conversely, history alterations generate anomalies in the pull-based development model [9], [23], potentially disrupting collaborative workflows.

*Best practices and guidelines:* Developers use alterations to maintain “clean” histories, in order to provide a clearer overview of project evolution and facilitate decision-making during code reviews and project management. Tools like, e.g., Githru [20] encourage squashing to improve clarity.

Cortés Ríos et al. [32] propose a taxonomy of workflows for collaborative software development, evaluating the pros and cons of approaches incorporating rebasing. Other researchers have focused on recommending processes for pull-based development models, analyzing metrics such as commit quantity per pull request [2] to guide merge decisions, and emphasizing that trustworthy software requires clean commit history [29].

Previous work looked into the challenges of resolving merge conflicts after rebases [17], [26], [34], offering tools and recommendations for their resolution in collaborative workflows.

*Tools for detecting history alterations:* Limited tools exist to help recover missing elements from commit history, either through side-effects or inference techniques. Abstract syntax tree (AST) based approaches [13] attempt to detect file renames with high accuracy, focusing on content rather than metadata alterations. Lavoie et al. [22] developed probabilistic methods to track file movements between versions, including addition, removal, and relocation, but they do not capture metadata alterations.

### B. Empirical analysis of history alterations

There are almost no empirical studies on history alterations, probably due in part to the practical difficulty of conducting them (cf. Section I). The most relevant prior work was conducted by Germán et al. [14], who subjected the official Linux kernel and its contributing repositories to continuous snapshots to better understand its development history. Their findings suggest that the actual development history of projects using DVCS can be irretrievably lost. While their study focused on alterations occurring across repositories, including rebasing and metadata changes, it did not examine alterations within individual repositories nor identify specific file-level changes, which we study at large-scale.

Previous work also identified the challenges of mining VCS histories for research purposes [4], [19], mentioning history rewrites as a significant concern. Flint et al. [12] studied timestamp anomalies in a small subset of SWH, suggesting that history alterations affect timestamp reliability, potentially compromising dataset integrity when mining software repositories. They did not study other kinds of alterations.

### C. Research gap

To our knowledge, no prior work has characterized, neither quantitatively nor qualitatively, the extent of history alterations in public code repositories, even less so at large-scale. The present work fills this gap. Furthermore, it provides a taxonomy to categorize history alterations, digs deeper in two recurrent patterns of alterations (secret suppression and retroactive license changes), and provides an automated tool to help developers who wish to be alerted in advance of the presence of history alterations in repositories of their interest.

## IX. CONCLUSION

We conducted the first large-scale study of altered version control (VCS) histories in public repositories, performing both quantitative and qualitative analyses of history alterations in 111 M (million) public repositories. We found evidence of history alterations in 1.22 M (million) repositories. The phenomenon is not negligible: due to it, downstream users of those repositories might have experienced disruption of their push/pull workflow, or might be currently using repositories that have been tampered with without their knowledge. We have shown that history alterations are neither confined to

unpopular repositories, nor to branches like pull request ones, where rewrites might be expected.

Applying a novel taxonomy for categorizing VCS history alterations, we discovered that most alterations involve changing the content of versioned files and directories. The next occurrence in terms of incidence are metadata-only alterations, involving changes to author names, commit messages, timestamps, etc. Conducting two targeted use cases we find that alterations are used to remove private information committed by mistake to repositories (e.g., private keys) and also to retroactively change the licensing of open source projects. As users and developers might want to audit projects for history alterations—and given the difficulty of doing so: history alterations are designed to leave no traces—we develop GITHISTORIAN, a novel tool that leverage the Software Heritage archive to automatically spot alterations and alert users, possibly as part of CI/CD pipelines.

In future work, we intend to build upon the general framework of this paper to investigate the relationship between history rewrites and supply chain attacks. That would require developing an approach to discern legitimate from malicious alterations and design automated detection systems for suspicious modification patterns. We also plan to look more closely at the timing of rewrites, as it can provide important insights into developer awareness and security practices. For instance, in the secret suppression scenario it would be interesting to distinguish early-stage removals, when the project was young and good security practices not mature yet, and late-stage removals which might be more worrisome. More generally for all rewrites one could distinguish between immediate corrections—where rewrites happen within minutes or hours of the rewritten commit(s)—and long-term exposures, where an history of commits persists for days, months, or more before rewrite.

#### ACKNOWLEDGMENTS

The authors would like to thank Olivier Barais for his invaluable suggestions on a preliminary version of this work.

#### REFERENCES

- [1] Jean-François Abramatic, Roberto Di Cosmo, and Stefano Zacchiroli. Building the universal archive of source code. *Communications of the ACM*, 61(10):29–31, September 2018.
- [2] Muhammad Ilyas Azeem, Sebastiano Panichella, Andrea Di Sorbo, Alexander Serebrenik, and Qing Wang. Action-based recommendation in pull-request development. In *ICSSP '20: International Conference on Software and System Processes, Seoul, Republic of Korea, 26-28 June, 2020*, pages 115–124. ACM, 2020.
- [3] Setu Kumar Basak, Lorenzo Neil, Bradley Reaves, and Laurie A. Williams. What are the practices for secret management in software artifacts? In *IEEE Secure Development Conference, SecDev 2022, Atlanta, GA, USA, October 18-20, 2022*, pages 69–76. IEEE, 2022.
- [4] Christian Bird, Peter C. Rigby, Earl T. Barr, David J. Hamilton, Daniel M. Germán, and Premkumar T. Devanbu. The promises and perils of mining git. In Michael W. Godfrey and Jim Whitehead, editors, *Proceedings of the 6th International Working Conference on Mining Software Repositories, MSR 2009 (Co-located with ICSE), Vancouver, BC, Canada, May 16-17, 2009, Proceedings*, pages 1–10. IEEE Computer Society, 2009.
- [5] Paolo Boldi, Antoine Pietri, Sebastiano Vigna, and Stefano Zacchiroli. Ultra-large-scale repository analysis via graph compression. In *SANER 2020: The 27th IEEE International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 2020.
- [6] Hudson Borges and Marco Túlio Valente. What's in a github star? understanding repository starring practices in a social coding platform. *J. Syst. Softw.*, 146:112–129, 2018.
- [7] Caius Brindescu, Mihai Codoban, Sergii Shmarkatiuk, and Danny Dig. How do centralized and distributed version control systems impact software changes? In Pankaj Jalote, Lionel C. Briand, and André van der Hoek, editors, *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 322–333. ACM, 2014.
- [8] Scott Chacon and Ben Straub. *Pro Git*. Apress, Berkely, CA, USA, 2nd edition, 2014.
- [9] Moataz Chouchen, Ali Ouni, Raula Gaikovina Kula, Dong Wang, Patanamom Thongtanunam, Mohamed Wiem Mkaouer, and Kenichi Matsumoto. Anti-patterns in modern code review: Symptoms and prevalence. In *28th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2021, Honolulu, HI, USA, March 9-12, 2021*, pages 531–535. IEEE, 2021.
- [10] Ludovic Courtès, Timothy Sample, Stefano Zacchiroli, and Simon Tournier. Source code archiving to the rescue of reproducible deployment. In *Proceedings of the 2nd ACM Conference on Reproducibility and Replicability, ACM REP 2024, Rennes, France, June 18-20, 2024*. ACM, 2024.
- [11] Roberto Di Cosmo and Stefano Zacchiroli. Software Heritage: Why and how to preserve software source code. In *Proceedings of the 14th International Conference on Digital Preservation, iPRES 2017, September 2017*.
- [12] Samuel W. Flint, Jigyasa Chauhan, and Robert Dyer. Pitfalls and guidelines for using time-based git data. *Empir. Softw. Eng.*, 27(7):194, 2022.
- [13] Akira Fujimoto, Yoshiki Higo, and Shinji Kusumoto. Towards accurate file tracking based on AST differences. In *28th Asia-Pacific Software Engineering Conference, APSEC 2021, Taipei, Taiwan, December 6-9, 2021*, pages 553–558. IEEE, 2021.
- [14] Daniel M. Germán, Bram Adams, and Ahmed E. Hassan. Continuously mining distributed version control systems: an empirical study of how linux uses git. *Empir. Softw. Eng.*, 21(1):260–299, 2016.
- [15] Hao He, Haoqin Yang, Philipp Burckhardt, Alexandros Kapravelos, Bogdan Vasilescu, and Christian Kästner. 4.5 million (suspected) fake stars in github: A growing spiral of popularity contests, scams, and malware. *CoRR*, abs/2412.13459, 2024.
- [16] Runzhi He, Hao He, Yuxia Zhang, and Minghui Zhou. Automating dependency updates in practice: An exploratory study on github dependabot. *IEEE Trans. Software Eng.*, 49(8):4004–4022, 2023.
- [17] Tao Ji, Liqian Chen, Xin Yi, and Xiaoguang Mao. Understanding merge conflicts and resolutions in git rebases. In Marco Vieira, Henrique Madeira, Nuno Antunes, and Zheng Zheng, editors, *31st IEEE International Symposium on Software Reliability Engineering, ISSRE 2020, Coimbra, Portugal, October 12-15, 2020*, pages 70–80. IEEE, 2020.
- [18] Jujutsu—a version control system. <https://jj-vcs.github.io/>, 2025. [Online; accessed 2025-05-25].
- [19] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. Germán, and Daniela E. Damian. An in-depth study of the promises and perils of mining github. *Empir. Softw. Eng.*, 21(5):2035–2071, 2016.
- [20] Youngtaek Kim, Jaeyoung Kim, Hyeon Jeon, Young-Ho Kim, Hyunjo Song, Bo Hyoung Kim, and Jinwook Seo. Githru: Visual analytics for understanding software development history through git metadata analysis. *IEEE Trans. Vis. Comput. Graph.*, 27(2):656–666, 2021.
- [21] Piergiorgio Ladisa, Henrik Plate, Matias Martinez, and Olivier Barais. Sok: Taxonomy of attacks on open-source software supply chains. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*, pages 1509–1526. IEEE, 2023.
- [22] Thierry Lavoie, Foutse Khomh, Ettore Merlo, and Ying Zou. Inferring repository file structure modifications using nearest-neighbor clone detection. In *19th Working Conference on Reverse Engineering, WCRE 2012, Kingston, ON, Canada, October 15-18, 2012*, pages 325–334. IEEE Computer Society, 2012.
- [23] Bohan Liu, He Zhang, Weigang Ma, Hongyu Kuang, Yi Yang, Jinwei Xu, Shan Gao, and Jian Gao. Mining pull requests to detect process

- anomalies in open source software development. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*, pages 194:1–194:13. ACM, 2024.
- [24] Jon Loelinger and Matthew MacCullogh. *Version Control with Git - Powerful Tools and Techniques for Collaborative Software Development: Covers GitHub, Second Edition*. O'Reilly, 2012.
- [25] Ralph C. Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *Advances in Cryptology - CRYPTO '87, A Conference on the Theory and Applications of Cryptographic Techniques, Santa Barbara, California, USA, August 16-20, 1987, Proceedings*, volume 293 of *Lecture Notes in Computer Science*, pages 369–378. Springer, 1987.
- [26] Hoai Le Nguyen and Claudia-Lavinia Ignat. An analysis of merge conflicts and resolutions in git-based open source projects. *Comput. Support. Cooperative Work.*, 27(3-6):741–765, 2018.
- [27] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. Backstabber's knife collection: A review of open source software supply chain attacks. In Clémentine Maurice, Leyla Bilge, Gianluca Stringhini, and Nuno Neves, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment - 17th International Conference, DIMVA 2020, Lisbon, Portugal, June 24-26, 2020, Proceedings*, volume 12223 of *Lecture Notes in Computer Science*, pages 23–43. Springer, 2020.
- [28] Philippe Ombredanne. Free and open source software license compliance: Tools for software composition analysis. *Computer*, 53(10):105–109, 2020.
- [29] Sachar Paulus, Nazila Gol Mohammadi, and Thorsten Weyer. Trustworthy software development. In Bart De Decker, Jana Dittmann, Christian Kraetzer, and Claus Vielhauer, editors, *Communications and Multimedia Security - 14th IFIP TC 6/TC 11 International Conference, CMS 2013, Magdeburg, Germany, September 25-26, 2013. Proceedings*, volume 8099 of *Lecture Notes in Computer Science*, pages 233–247. Springer, 2013.
- [30] Antoine Pietri, Diomidis Spinellis, and Stefano Zacchiroli. The software heritage graph dataset: public software development under one roof. In Margaret-Anne D. Storey, Bram Adams, and Sonia Haiduc, editors, *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada*, pages 138–142. IEEE / ACM, 2019.
- [31] Solal Rapaport, Laurent Pautet, Samuel Tardieu, and Stefano Zacchiroli. Replication package for: Altered histories in version control system repositories: Evidence from the trenches. <https://doi.org/10.5281/zenodo.15544257>, 2025. Archived in Software Heritage with SWHID swh:1:rev:3fbf8ceda6256d3d32ed2081351e0677089cf29b.
- [32] Julio César Cortés Ríos, Suzanne M. Embury, and Sukru Eraslan. A unifying framework for the systematic analysis of git workflows. *Inf. Softw. Technol.*, 145:106811, 2022.
- [33] ScanCode-Toolkit documentation. <https://scancode-toolkit.readthedocs.io/>, 2025. [Online; accessed 2025-05-29].
- [34] Chaochao Shen, Wenhua Yang, Minxue Pan, and Yu Zhou. Git merge conflict resolution leveraging strategy classification and LLM. In *23rd IEEE International Conference on Software Quality, Reliability, and Security, QRS 2023, Chiang Mai, Thailand, October 22-26, 2023*, pages 228–239. IEEE, 2023.
- [35] Stack Overflow. Stack overflow developer survey 2022. <https://survey.stackoverflow.co/2022/#section-version-control-version-control-systems>, 2022.