# Chasing One-day Vulnerabilities Across Open Source Forks

Romain Lefeuvre[1], Charly Reux[1], Stefano Zacchiroli[2], Olivier Barais[1], Benoit Combemale[1]

[1]University of Rennes, France {romain.lefeuvre, charly.reux, olivier.barais, benoit.combemale}@irisa.fr
[2]LTCI, Télécom Paris, Institut Polytechnique de Paris, France stefano.zacchiroli@telecom-paris.fr



***Abstract*—Tracking vulnerabilities inherited from third-party open-source components is a well-known challenge, often addressed by tracing the threads of dependency information. However, vulnerabilities can also propagate through *forking*: a repository forked after the introduction of a vulnerability, but before it is patched, may remain vulnerable in the fork well after being fixed in the original project. Current approaches for vulnerability analysis lack the commit-level granularity needed to track vulnerability introductions and fixes across forks, potentially leaving *one-day vulnerabilities* undetected.**

**This paper presents a novel approach to help developers identify one-day vulnerabilities in forked repositories. Leveraging the global graph of public code, as captured by the Software Heritage archive, the approach propagates vulnerability information at the commit level and performs automated impact analysis. This enables automatic detection of forked projects that have not incorporated fixes, leaving them potentially vulnerable.**

**Starting from 7162 repositories that, according to OSV, include vulnerable commits in their development histories, we identify 2.2 M forks, containing at least one vulnerable commit. Then we perform a strict filtering, allowing us to find 356 ⟨vulnerability, fork⟩ pairs impacting active and popular GitHub forks, we manually evaluate 65 pairs, finding 3 high-severity vulnerabilities, demonstrating the impact and applicability of this approach.**

## I. Introduction

Open source software plays a critical role in the global digital infrastructure [15]. Its popularity has increased greatly over the past decades, thanks to its unrivaled ease of code reuse. There are two main mechanisms for the reuse of open-source software [3]: *dependencies* and *"forks"*. The dependency mechanism involves calling into the code of a separate, freely licensed software product via its standard API. Forking means reusing the source code, and the entire development history of an existing open-source project, to establish a separate development trajectory, resulting in a distinct software product [38].

Forking serves multiple purposes: facilitating large-scale development through a GitFlow-like approach [33], addressing industrial or interpersonal conflicts within the community, or working as a reuse strategy to extend functionality without having to integrate new ideas into the original project [24], [39]. This can be observed empirically too: more than 40% of repositories hosted on GitHub are forks[1].

The security of the global open-source ecosystem attracted significant attention in recent years, particularly in the wake of high-profile software supply chain attacks on specific dependencies [16], [10]. Most research in this area focused on the challenges associated with secure dependency management [13]. The risk of vulnerability propagation through dependencies is well documented in the literature, emphasizing the need for thorough audits to identify vulnerabilities and implement mitigation measures, such as upgrading to a secure version (or applying the corresponding patches) as quickly as possible. The security implications of fork-based reuse have received comparatively less attention. This is in stark contrast with the fact that fork-based reuse is both widespread and foundational in major open-source projects. For instance, the Linux kernel has numerous forks, including prominent derived operating system kernels such as Android.

*a) Problem statement:* This paper focuses on the under-explored topic of "cross-fork" vulnerability propagation in the open-source ecosystem. Forks share parts of a common code history, and a vulnerability identified in one fork is likely to affect others, depending on when forking happened and how changes in the original ("upstream") project are incorporated into ("downstream") forks. For example, if a downstream project is forked in between an upstream commit that introduces a "high severity" vulnerability and the corresponding commit that fixes it (assuming it exists), the downstream project will remain vulnerable until the fixing commit is merged back there. From the moment the vulnerability is discovered and the fix commit is integrated, the downstream project is affected by a so-called *one-day vulnerability*: a known, but unpatched vulnerability.

The failure to propagate vulnerability information at the granularity of commits from upstream projects to downstream forks increases the number of one-day vulnerabilities, and with them the risks for end users (of forks). Given the current lack of approaches and tools to track cross-fork vulnerability propagation, fork maintainers are tasked with manually tracking upstream advisories and assessing, for each vulnerability, whether it applies to them and if the corresponding fixes have been integrated. Fork users are also impacted, due to the fact that only major, well-known forks are tracked by vulnerability databases: users of the myriad other forks will generally ignore that they are impacted by upstream one-day vulnerabilities inherited via forking.

Addressing this issue requires: (1) identifying forks, ideally at a global scale; (2) keeping track of which commits introduce

[1]Analysis based on an export of Software Heritage as of April 3, 2024

and fix security vulnerabilities; and (3) propagating the information about which commits are vulnerable, from upstream projects to downstream forks. This challenge is compounded by the diverse and heterogeneous nature of software forges (GitHub, GitLab in multiple self-hosted instances, ...) and the use of different version control systems (Git, Mercurial, Subversion, ...).

*b) Contributions:* We design an automated approach for propagating commit-level vulnerability information from a project development history to its forks using a unified model at the global scale of all publicly available commits. We implement the proposed approach using the commit graph from Software Heritage, which is the largest public archive of software source code and associated development history. We show that this approach can be leveraged in software development workflows (e.g., in a CI pipeline) to evaluate whether forks of interest (e.g., those found in our own software supply chain) are affected by one-day vulnerabilities.

Specifically, we address the following research questions:

**RQ1: Can we leverage the global commit graph to track vulnerability introductions and fixes throughout the open-source ecosystem?**

**RQ2: Using commit-level information about vulnerability introductions and fixes, can we identify unfixed vulnerabilities in real-world forked open-source projects?**

To answer these research questions, we conducted a large-scale empirical study involving 7162 upstream repositories associated to known vulnerabilities, as indexed by OSV (see Section II-B). We identified 2.2 M forks of these projects and then applied the proposed approach to identify those among them potentially affected by one-day vulnerabilities. We then applied a strict filtering on the identified ones, manually verified that they were in fact vulnerable, informed their maintainers, and assessed their responses. This manual evaluation allowed us to easily spot 44 potential and 3 confirmed real one-day vulnerabilities with high severity, strengthening the relevance of our approach.

*c) Paper structure:* The remainder of the paper is organized as follows: Section II provides the background for this study. Section III outlines our methodology and approach for propagating vulnerability information throughout the fork ecosystem. Section IV Leverages the propagation of vulnerability information to identify unpatched forks. Section V presents tooling integration in the software development process. Section VI discusses the implications of our findings on development practices and the threat to validity. Section VII reviews related work on vulnerability analysis using version control systems (VCS). Finally, Section VIII concludes with a discussion of our findings.

## II. Background

### A. Forking

*Forking* is an open-source practice that consists in creating a new source code repository from an existing one, usually preserving prior development history. Traditionally, "forking" meant splitting development efforts to bring a project in a different technical direction than the one meant by the current maintainers. These kinds of fork, which result in the creation of new software, are referred to as *hard forks*.

With the advent of collaborative code hosting platforms [7], a new kind of fork has emerged, called *social fork*: contributors create personal forks of the projects they want to contribute to, make changes there, and then propose them for integration to upstream maintainers via mechanisms such as pull requests [39]. Social forks tend to be ephemeral and cease to be relevant after contributions are integrated, but in some cases they can grow their own user base and eventually become more popular than the original project, blurring the lines between hard and soft forks.

In this paper, to *discover forks* of a project, we rely on the structural properties of the global commit graph produced by distributed version control systems (DVCSs) like Git. Specifically, we rely on the definition of *shared commit fork* from Pietri et al. [28]: "*A repository $B$ is a* shared commit fork *of repository $A$, [...] if it exists a commit $c$ contained in the development histories of both $A$ and $B$.*" This notion allows detecting more forks than those known to specific platforms such as GitHub. It can additionally detect as forks repositories pushed independently to a development platform (as opposed to only those created clicking on "Fork this repository" there), as well as "cross-forge" repositories hosted on different platforms (e.g., identifying one GitLab project as a fork of another hosted on GitHub).

To *identify one-day vulnerabilities*, we focus on forks that have users, according to platform-specific popularity metrics (see Section IV for details). We can hence identify vulnerabilities in both hard forks that diverged from upstream or social forks containing changes not yet integrated upstream, which are gaining traction with users.

Different development approaches can propagate code changes between related forks. Some approaches, like those based on `git merge`, mesh commits coming from different forks, retaining the identity of individual commits. Other approaches, like those based on `git cherry-pick`, "import" commits across forks without retaining commit identities, but can leave explicit traces in metadata like commit messages.

### B. OSV (Open Source Vulnerabilities)

*OSV (Open Source Vulnerabilities)*[2] is a project by the OpenSSF foundation consisting of two parts: (1) a standard format[3] to describe vulnerabilities that affect open-source products, and (2) a public database of vulnerability advisories encoded in that format, which aggregates security information from more than 20 software ecosystems.

The OSV format associates vulnerabilities to *version ranges* within software development histories. Each range describes a sequence of *events*, e.g., the *introduced* event references the version where the vulnerability was introduced, while *fixed* references the version where it has been corrected. Versions

[2]https://osv.dev

[3]https://ossf.github.io/osv-schema

in events can be specified in various ways, including version numbers and Git commit SHA1 identifiers. Reasoning at commit level enables more fine-grained analyses than version numbers and is particularly useful when studying forks. In the remainder of the paper, we will only consider vulnerabilities recorded by OSV whose ranges are described using Git ranges.

The OSV database backs an API that can answer queries like *"Is version X of product Y affected by any known vulnerability?"*. In order to answer queries about specific commits, OSV periodically crawls the repositories of all projects that have ever been associated to at least one vulnerability. It then evaluates an algorithm (defined as part of the OSV standard) to decide whether each commit in the project history is vulnerable or not, and maintains up-to-date a $commit \rightarrow^* vulnerability$ relationship, used to answer queries. Crucially, OSV only knows about commits in upstream projects and would, in the general case, produce false negative answers when queried about commits that are only found in downstream forks.

### C. Software Heritage

Software Heritage (SWH) [6][4] is the largest public archive of software source code and associated development history. It archives code from more than 300 million projects coming from development forges (GitHub, GitLab, etc.) and package manager repositories, resulting in more than 20 billion files and 4 billion commits. The SWH data model is a fully-deduplicated Merkle directed acyclic graph (DAG) structure, with nodes representing common source code artifacts, including (Git) commits and releases. The SWH graph is the largest current approximation of the global commit graph and links public code commits across unrelated development platforms.

Each node of the SWH graph is identified by unique and persistent "SoftWare Hash IDentifiers" (SWHID).[5] For commit objects, SWHIDs are compatible with Git identifiers: they can be mutually derived from one another. The SWH graph, formed by around 50 billion nodes and 0.5 trillion arcs, can be loaded into main memory using a compressed representation [29] that allows performing scale-up analyses that would be prohibitively costly in other representations.

The *deduplication* property of the SWH Graph makes it particularly useful for studying project forks. For instance, forks hosted on different platforms, as well as forks independently pushed to the same platform, will be part of the same connected component of the graph, allowing to study forks globally as previously done in [28]. Doing so analyzing repositories separately would be much more difficult and could result in overlooking forks.

## III. Tracking vulnerable commits across forks

We now describe the proposed methodology for systematically tracking commits that introduce and fix known vulnerabilities across all downstream forks of a given upstream repository. The approach is designed to operate at global scale, enabling the identification of vulnerability-related code changes in distributed software ecosystems.

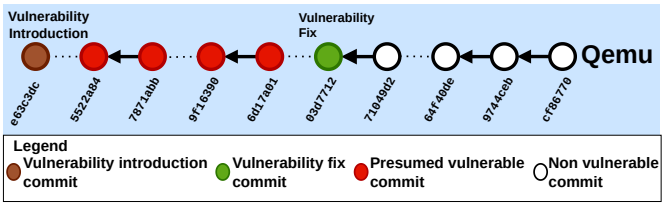



Fig. 1: QEMU Git history: local repository-level analysis. (Note that commits are built bottom-up on previous ones. Most recent commits are graph roots, shown here on the right.)

Section III-A uses a real-world case study to illustrate the risks associated with global propagation of one-day vulnerabilities via forks. Section III-B provides a formal description of the algorithm designed to trace the spread of vulnerabilities on a global scale. Section III-C outlines the experimental methodology used to answer RQ1. Section III-D discusses the findings derived from this empirical investigation.

### A. Reasoning on the global commit graph

Current approaches for identifying specific commits affected by a vulnerability rely on local (repository-level) analysis. For example, in the case of OSV.dev, the Git history of each repository associated in the past with at least one vulnerability is cloned and analyzed to detect vulnerable commits. A database of vulnerable commits is then used as backend for an API that allows to lookup vulnerabilities associated to a given commit. Forks of open source projects share parts of a common development history with their parent project, potentially inheriting unfixed vulnerabilities. Repository-local analyses induce a "blind spot" problem that can result in one-day vulnerabilities (i.e., known but unpatched vulnerabilities) being overlooked in project forks.

Let us look at a real case of an unpatched fork involving a high-criticality vulnerability(CVSS base score: 7.8 HIGH), identified as CVE-2019-13164,[6] impacting PANDA, a QEMU fork. QEMU [2] is a well-known open-source generic machine emulator and virtualizer, whereas "PANDA is an open-source Platform for Architecture-Neutral Dynamic Analysis [...] built upon the QEMU whole system emulator."[7] PANDA started as a hard fork of QEMU and then evolved independently of it for over a decade, rarely integrating commits from the upstream project. Owing to their initially shared developmental history, PANDA inherits vulnerabilities present in QEMU at the time of the fork, including CVE-2019-13164. The vulnerability is considered to be already present in the first Git commit of QEMU and was later fixed in commit `03d7712`.

Figure 1 illustrates the presumed vulnerable commits in QEMU Git history, identified with a local analysis. Figure 2 depicts a portion of the PANDA Git history, which does not

[4]https://www.softwareheritage.org
[5]https://swhid.org
[6]https://nvd.nist.gov/vuln/detail/CVE-2019-13164
[7]https://panda.re

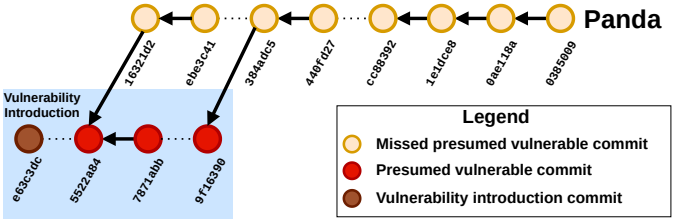



Fig. 2: Propagation of potentially vulnerable commits from upstream QUEMU repository to its downstream PANDA fork.

include the fix commit `03d7712`. The portion of the history in the blue box is shared with QEMU.

The commits represented in this section of the graph are identical to those in the QEMU repository, as indicated by their matching Git identifiers. The vulnerability status of these commits is effectively assessed by examining them in the context of the QEMU Git repository. In contrast, the subsequent portion of PANDA's Git history contains vulnerable commits that are exclusive to PANDA and, therefore, lie entirely beyond the scope of QEMU's local analysis. Neither is the PANDA repository analyzed separately: the project is not referenced in the associated vulnerability report, leading to an overlooked one-day vulnerability. Note that the *new* presumed vulnerable commits in PANDA history have commit identities (as SHA1 identifiers) missing from the upstream QEMU repository; therefore they cannot be spotted as potentially vulnerable based on commit identities, because they will not be found in vulnerability databases.

Handling these cases requires a two-step process: first, identify all forks that may be impacted by a vulnerability, and second, analyze each of them. However, identifying all forks of a repository requires the systematic examination of open-source repositories hosted across multiple code-hosting platforms. Even if a comprehensive list of public forks of a given vulnerable upstream repository could be obtained, analyzing each fork individually would lead to significant redundancy in the effort. Forking a repository is cheap, whereas vulnerability testing is resource-intensive.

Figure 3 recast our QEMU/PANDA example in the context of a global model of commits, like the one provided by SWH (see Section II-C). In such a model, repository histories are deduplicated and hence linked together by the identities of shared commits. The list of vulnerable commits can now be identified by listing commit nodes between introduction and fix commits. As a result, all the commits in PANDA that were not detectable as vulnerable by separately analyzing the two repositories can now be spotted: by propagating the information that commit `e63c3dc` introduced the vulnerability and that commit `03d7712` fixed it, we can detect all PANDA commits starting at commit `16321d2` as potentially vulnerable.

### B. Commit labeling

To implement in practice this idea, we need to "label" each commit in the global commit graph with the set of known vulnerabilities affecting it. We will obtain these labels by

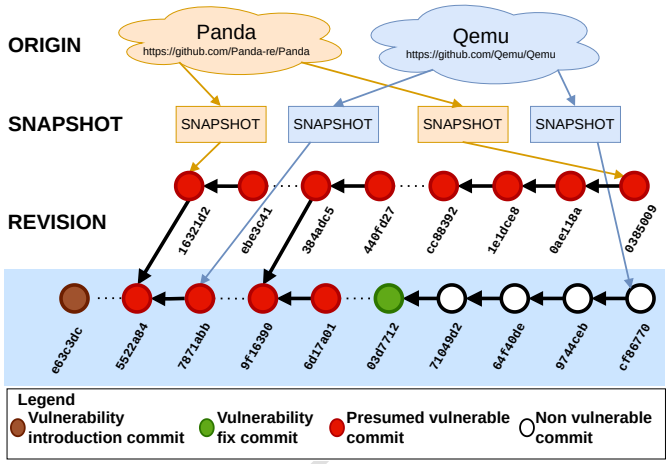



Fig. 3: Running example on SWH model

propagating commit-level information about which commits introduce or fix vulnerabilities from upstream repositories to downstream forks. We detail below the main parts of an algorithm that does so. They correspond to a formalization of the evaluation semantics of OSV,[8] which is applied in that context to individual repositories, and generalized here to the global commit graph.

*a) State:* We declare the following variables:

- $\texttt{commit\_graph} = (C, E)$: A DAG of Commits $C$ with Edges $E \subseteq C \times C$ such that each edge $(c_2, c_1) \in E$ indicates that $c_2$ is a (recent) commit referencing a (previous) commit $c_1$.[9]
- $\texttt{vuln\_ranges} = \{R_1, R_2, \ldots, R_n\}$, with each $R_i$ a vulnerability range.
- With each range $R$ a record of type:

$$R : \{\texttt{intro} \subseteq C, \texttt{fixed} \subseteq C, \texttt{limit} \subseteq C, \texttt{last} \subseteq C\}$$

where the different events are defined as follows, according to OSV semantics (see Figure 4):

- `intro`: the commit(s) that introduce a vulnerability.
- `fixed`: commit(s) where the vulnerability was fixed.
- `limit`: similar to fixed, but restricts the propagation of vulnerable commits to the branch where the limit commit is located.
- `last`: last affected commit, i.e., last vulnerable commit along a history branch.

*b) Global initialization:* The global mapping from commits to vulnerability ranges is maintained in: $\texttt{commit\_to\_vuln\_range} \leftarrow \emptyset$

*c) Iteration over range:* The main loop of the algorithm fills the global mapping by iterating on all vulnerability ranges.

$$\forall R \in \texttt{vuln\_ranges} :$$

The rest of the algorithm below happens within this loop.

[8] https://ossf.github.io/osv-schema/#evaluation

[9] In Git terminology $c_1$ is called the *parent* commit of $c_2$, because it comes earlier, but in the commit graph, it is $c_2$ that points to $c_1$.

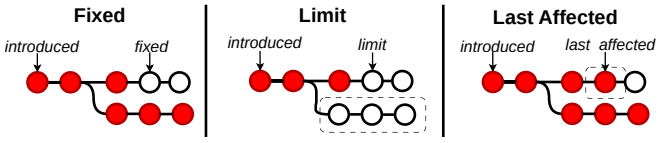



Fig. 4: OSV events and their semantics: commits filled in red are considered vulnerable, empty commits are not. Differences with fixed event are dashed.

*d) Local initialization:* For each range, we maintain a mapping from traversed commits to whether they are patched or not and use a stack to visit the subgraph starting at the introduction commit.

$$\texttt{patched\_commits} :\leftarrow \{\}$$
$$\texttt{stack} \leftarrow R.\texttt{intro}$$
$$\forall c \in R.\texttt{intro}, \quad \texttt{patched\_commit}(c) \leftarrow \texttt{False}$$

*e) Traversing the commit graph:* We perform a DFS visit from all *introduced commits* to discover all reachable commits and evaluate their vulnerability status. The status of all parents is not necessarily known when a commit is analyzed, hence it needs to be dynamically updated during visit. For example, if a child is first processed from a vulnerable parent and later encountered again from a patched parent, its status is updated, and the status of its children recomputed. In the case of merge commits, we apply a conservative policy of "fix propagation": a merge commit is considered to be no longer vulnerable if at least one of its parent commits is fixed.

While $\texttt{stack} \neq \emptyset$ :
  $c \leftarrow \texttt{stack.pop()}$
  $\forall (c', c) \in \texttt{E}$ :
    $\texttt{is\_patched}(c') \leftarrow \neg(c' \in R.\texttt{intro} \wedge ($
      $c' \in R.\texttt{fixed} \vee c' \in R.\texttt{limit}$
      $\vee \texttt{patched\_commit}(c')$
      $\vee \texttt{patched\_commit}(c)$
      $\vee c \in R.\texttt{last})$
    If $\texttt{patched\_commit}(c') \neq \texttt{is\_patched}(c')$ :
      $\texttt{patched\_commit}(c') \leftarrow \texttt{is\_patched}(c')$
      $\texttt{stack} \leftarrow \texttt{stack} \cup \{c'\}$

*f) Limit filtering:* For ranges with limit events, we drop all vulnerable commits not on the relevant history branch.

If $R.\texttt{limit} \neq \emptyset$ :
  $\texttt{patched\_rev\_filtered} \leftarrow \emptyset$
  $\texttt{stack} \leftarrow R.\texttt{limit}$
  While $\texttt{stack} \neq \emptyset$ :
    $c \leftarrow \texttt{stack.pop()}$
    $\forall (c, p) \in E$ :
      If $\neg\texttt{patched\_commit}(p)$
        $\texttt{patched\_rev\_filtered}(p) \leftarrow \texttt{False}$
      $\texttt{stack} \leftarrow \texttt{stack} \cup \{p\}$
  $\texttt{patched\_commit} \leftarrow \texttt{patched\_rev\_filtered}$

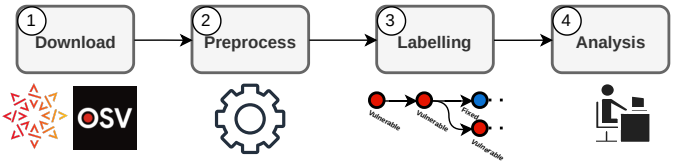



Fig. 5: Experimental protocol steps

*g) Updating result:* Finally, the global vulnerability status of each commit is updated for the current range.

$\forall c \in C,$
  if $\neg\texttt{patched\_commit}(c)$
    $\texttt{commit\_to\_vuln\_range}(c)$
      $\leftarrow \texttt{commit\_to\_vuln\_range}(c) \cup \{R\}$

The final output is a mapping from commits to OSV ranges, and hence associated vulnerabilities. Note that, for ease of presentation, the algorithm presented above is a simple multi-pass algorithm, doing one pass per range. An alternative, more efficient single-pass implementation is also possible, but was not needed in practice in our experience.

### C. Experimental protocol

To answer RQ1, we designed and executed an experimental protocol to propagate OSV vulnerability information to the global commit graph from SWH. The experimental protocol consists of the 4 steps shown in Figure 5. First, we retrieve: (1) a recent export of the global commit graph from Software Heritage[10] and (2) export vulnerability information from OSV[11] as of 2024-11-27.

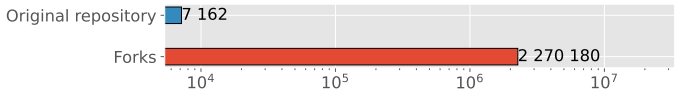
Second, we clean up inconsistencies from OSV data, such as multiple event types that point to the same commit or formatting issues. If an inconsistency is detected within a range, we exclude the range. We also exclude ranges having events that refer to commits not present in the SWH graph (e.g., because they are more recent than our dataset).

We also perform some data-augmentation transformations. For ranges with an introduction event pointing to "0", the vulnerability was present since the beginning of the project commit history; we hence identify the leaf commits of the relevant repository and complete the ranges with their identities. We also handle cherry-picking of event commits across branches (possibly belonging to different forks); we detect commit messages that explicitly describe cherry picks via the default Git message `"cherry picked from commit..."`. We add detected cherry-picked commits as additional events for the same range.

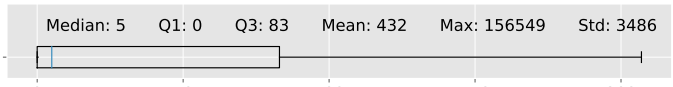
Third, we apply the propagation algorithm of Section III-B to the SWH commit graph and OSV ranges (after data augmentation). We obtain as a result a mapping from all SWH commits to OSV vulnerability ranges.

[10]We used version 2024-05-16, documented at https://docs.softwareheritage.org/devel/swh-dataset/graph/dataset.html.

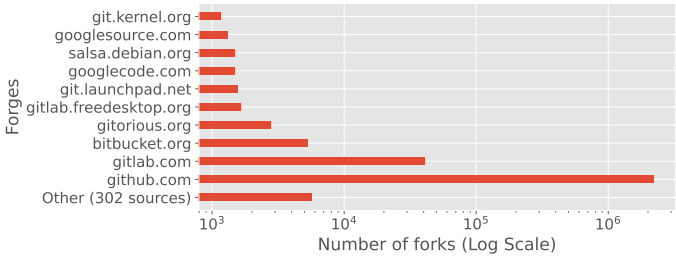
[11]https://google.github.io/osv.dev/data/#data-dumps




(a) Number of forks compared to upstream repositories

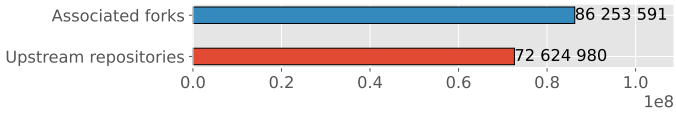



(b) Distribution of the number of forks, by upstream repository

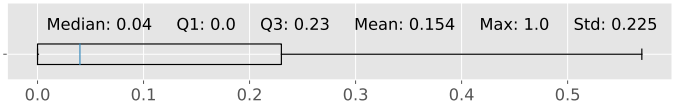



(c) Number of forks by forge

Fig. 6: Forks of OSV referenced repositories, having at least a new vulnerable commit




(a) Number of new vulnerable commits in forks compared to upstream repositories




(b) Ratios distribution of new vulnerable commits in forks over all vulnerable commits, per vulnerability (range)

Fig. 7: New vulnerable commits in forks

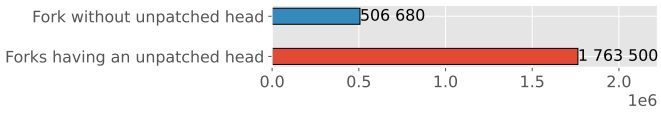



Fig. 8: Forks having a presumed vulnerable commit in their history : proportion of fork having an unpatched head

As fourth and last step, we conduct an in-depth analysis of popular potentially vulnerable unpatched forks, as detailed in Section IV-C.

For the automated part of the protocol, and in particular step (3), we run our experiments on a server with the following characteristics: 16 × 32 GiB DDR4 memory, 3200 MHz; 6 TiB of SSD storage; 2× AMD EPYC 7543 32-core CPUs, 2.8 GHz; Ubuntu 22.04.

### D. Potentially vulnerable commits: descriptive statistics

Figure 6 shows that, after commit labeling (step 3), starting from an initial set of 7162 repositories referenced by OSV, we could identify 2.2 M (million) forks containing at least one presumed vulnerable commit. On average, each initial repository impacts 432 forks. The first quartile of 0 and median of 5 suggest that a few upstream repositories have a large amount of vulnerable forks. Although the majority of vulnerable forks are hosted on GitHub, find potentially vulnerable commits spread across 312 distinct forges (by Internet domain name), confirming that vulnerable forks tend to spread around: platform-specific solutions are not enough to track them all.

As shown in Figure 7 we identified 158.9 M commits as potentially vulnerable, 86.6 M of which can only be found (by commit identity) in fork histories, accounting for 54.5% of all potentially vulnerable commits. Considering individual vulnerability ranges, we observe that on average 15% are presumed vulnerable commits only on forks history. The standard deviation of 22% indicates high dispersion. In 50% of the ranges, only 4% of vulnerable commits exist only in forks history, while 25% of the ranges have 22% of vulnerable commits only in forks history.

**RQ1:** The potential impact of cross-fork vulnerability propagation is high: starting from 7162 repositories, we find more than 2.2 M forks on 312 forges, containing at least one potentially vulnerable commit in their history.

What remains to be seen is how many, among these *potential* vulnerabilities according to commit-level information, are *actual* one-day vulnerabilities. We delve into this next.

## IV. One-day vulnerable forks

We introduced an approach to detect forks with new and potentially vulnerable commits, that are not present in the upstream repository and cannot be detected by traditional methods. We now look into these commits, to see if they correspond to *actual* vulnerabilities in real-world forked open-source projects (as per RQ2).

To identify real-world one-day vulnerabilities, we first identify *presumed unpatched forks*, i.e., forks that have one of their branch heads affected by a vulnerability that has already been patched in a repository referenced by OSV. As a result, we exclude forks that contain vulnerable new commits in their histories, but fixed them since. We identify 1.7 M presumed unpatched fork, representing 77.7% of all forks previously identified as containing at least one vulnerable commit in their histories (Figure 8).

The remainder of this section outlines the experimental protocol used to identify real-world one-day vulnerabilities. It presents the filtering process applied to the set of potentially vulnerable forks to derive a subset that is amenable to human analysis (Section IV-A). Then, it describes the validation procedure used to assess the presence of one-day vulnerability (Section IV-B). Section IV-C presents the results of this experimental study, summarizing the main findings.

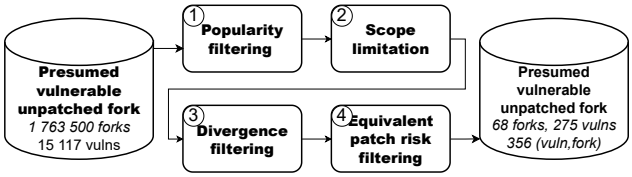



Fig. 9: Strict filtering methodology

### A. Strict filtering process

Presumed unpatched forks are forks that have a commit in their history that introduces a vulnerability without having the available fix commit. The absence of a vulnerability fix does not necessarily imply that the fork is actually affected. The fork may have diverged enough from the original codebase and no longer be affected by the vulnerability. Another reason could be that the vulnerability has been fixed via an alternative approach than the upstream one. Finally, many of the forks we identified are stale social forks used for pull/merge requests that have no actual users. To identify relevant one-day vulnerabilities given all these constraints, we applied the strict filtering methodology shown in Figure 9.

*1) Popularity filtering:* We initiated the analysis by implementing a filtering stage to identify social and/or hard forks with distinct user bases. To approximate this criterion, we employed popularity metrics, specifically the number of stars and forks provided by GitHub. The filtering criteria were defined as follows: forks hosted on GitHub with more than 100 stars and 10 forks. Consequently, the subsequent filtering process focused exclusively on the forks hosted on GitHub. The decision was influenced by the fact that most of the detected forks were hosted on this platform. Additionally, GitHub provides useful social metadata, including metrics like the number of stars and forks.

*2) Scope limitation:* We limit the scope of the analysis in various ways. First, at the vulnerability level, we only retain highly critical vulnerabilities (CVSS score $> 7$), and only those that have a fix event in OSV, since we require a precise reference to the patch to evaluate whether or not a fork applies it. Second, we refine the dataset by excluding pairs of $\langle fork, vulnerability \rangle$ where the fork was archived or where the affected head commit was not located on the main branch. To perform this filtering and the following steps, all repositories are cloned locally for git history analysis.

*3) Divergence filtering:* We exclude forks that have diverged too much from upstream. To evaluate this, we look at the file(s) impacted by the fix commit. If at least one of the affected files is not present in the fork repository, we exclude the fork. (While executing this filtering step we track file renames properly using `git log follow --find-rename`.)

*4) Equivalent patch risk filtering:* The last step is designed to filter out pairs of $\langle fork, vulnerability \rangle$, with a significant chance that an alternative patch has been applied in the fork. First, we analyze fork histories to exclude those containing commits that explicitly mention the relevant CVE identifier in their commit messages. Additionally, we perform a more thorough detection of cherry-picked commits than what was performed on the global graph. We compute the git patch id[12] of each commit and use it to detect cherry-picked fixes.

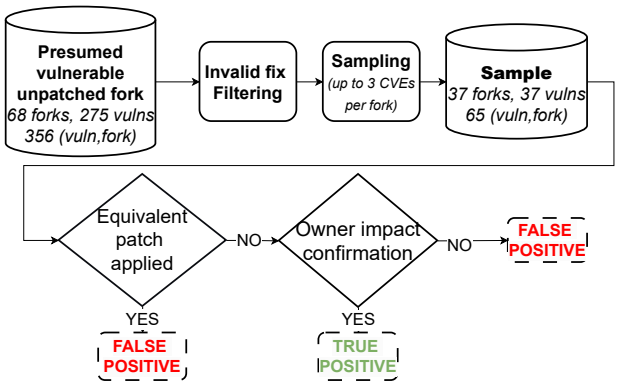



Fig. 10: Manual analysis protocol (adjusted from maintainers' responses)

At the end of this strict filtering we obtain 68 presumed-vulnerable forks and 275 unique vulnerabilities.

### B. Vetting of unpatched forks

To evaluate whether our analysis effectively facilitates the detection of unpatched forks, we manually evaluated the presumed unpatched forks obtained after filtering. For each case we manually verify the absence of the fix in the fork and then contact the fork maintainers for validation. The vetting protocol, which follows these steps, is depicted in Figure 10 and detailed below.

*1) Evaluation of invalid CVE fixes:* The first step consists of an evaluation of the data quality of the vulnerability report used in input of our approach. CVE databases are growing massively and rely heavily on unstructured data for which quality and consistency are a common concern [8]. Although based on a standardized format, the OSV database, which aggregates multiple vulnerability databases, is not exempt from data quality issues. For instance, many "fixed" commits do not contain vulnerability patches, but only version updates on configuration files. To assess whether a fix has not been applied in a fork, the "fixed" commits in the vulnerability report should effectively point to code modification. Thus, we manually reviewed and filtered 136 vulnerabilities that had an incorrect fix commit out of 275, resulting in 168 pairs of $\langle fork, vulnerability \rangle$ with 37 unique forks and 136 unique vulnerabilities.

*2) Sampling:* As manually analyzing 168 pairs of $\langle fork, vulnerability \rangle$ was still not feasible, we performed a sampling. For each of the 37 forks, we randomly sampled up to 3 vulnerabilities. This sampling resulted in 65 pairs of $\langle fork, vulnerability \rangle$, because some forks contained only one or two vulnerabilities.

[12]https://git-scm.com/docs/git-patch-id

*3) Equivalent patch application:* An equivalent patch is a patch contained in a commit having a different commit id and patch id than the original fix commit. For each of the 65 pairs of $\langle fork, vulnerability \rangle$, an author evaluates whether an equivalent patch has been applied, as follows:

- Reading of the CVE description.
- Reading of the patch modification on the patch commit.
- For each modified file, the modifications are checked on the presumed unpatched fork.
- If an equivalent patch is found, the pair $\langle fork, vulnerability \rangle$ is considered a false positive.

As a result of this stage, 21 pairs $\langle fork, vulnerability \rangle$ were considered false positives, due to the application of equivalent patches.

*4) Maintainer notification:* We contacted the maintainers of remaining forks identified as potentially vulnerable to one-day vulnerabilities. This constitutes both responsible disclosure of potential security issues that we identified, and provides the opportunity to ultimately validate our findings by the most relevant experts there is: the maintainers. Based on the obtained maintainer answers, we ultimately classify presumed vulnerable forks as:

- *False positive*: the fork is not vulnerable for reasons we could not identify ourselves, e.g., the vulnerability was not applicable to it or the patch was applied via means we did not detect;
- *True positive*: the fork is actually vulnerable.

### C. Results

By executing the protocol above on the initial 65 vetted pairs $\langle fork, vulnerability \rangle$, we excluded 21 and retained 44 as candidates for maintainer confirmation. Among the candidates for confirmation, here are a few notable examples:

- `sonyxperiadev/kernel`: the Linux kernel fork by the Sony corporation, for Xperia Android devices.
- `panda-re/panda`: a platform for Architecture-Neutral Dynamic Analysis (our initial example).
- `reMarkable/linux`: the Linux kernel fork for re-Markable eInk tablets.

We contacted the maintainers of 23 forks, for 38 pairs. (We excluded 6 pairs because, at the time of final due diligence before contacting maintainers, they were associated to forks explicitly documented as deprecated or not-for-use, including some meant for educational purposes only.) We received 7 answers: 3 confirmation of vulnerabilities and 4 false positives. For the remaining ones we are awaiting confirmation.

We received two types of responses: (1) from active communities where the maintainers of the repositories quickly answered and fixed the vulnerability, (2) from less active forks, with communities lacking maintainer resources and struggling to keep up with upstream. An example of the first type of answer is for `github.com/sonyxperiadev/kernel`, vulnerable to CVE-2021-45485 and CVE-2021-4154, whose maintainers fixed the vulnerability within days.

The majority of received responses were of the second type, with maintainers acknowledging the vulnerable status of their fork, but lacking time to do something about it.

Our RQ2 conclusion is then:

> **RQ2:** Commit-level information about vulnerability introduction and fixes is effective in supporting the identification of downstream forks affected by one-day vulnerabilities, as exemplified by the `PANDA` and `Xperia` case. More generally, among the 37 repositories initially detected as vulnerable (from an initial sample of 68 popular forks), **26 forks remained classified as vulnerable after manual vetting, and 23 have been notified. We received 7 answers, confirming 3 one-day vulnerabilities**.

## V. Integration in software development processes

Although we demonstrated the applicability of the approach in a large-scale empirical experiment, the practicality of its integration into real-world software development processes is a requirement to actually improve security for fork users. We analyze it in the following section.

### A. Use Cases

The proposed approach can be applied in two distinct contexts. Initially, it can assist fork maintainers in recognizing vulnerabilities that have been reported in other forks and are also relevant to their own. When vulnerabilities are identified, the approach also helps to identify which patches should be applied to fix them. Second, in the context of traditional dependency-based audits, this approach can be integrated to warn about dependencies that are themselves forks and have not integrated upstream vulnerability fixes.

### B. Case study: Git submodules and Go dependencies

As a concrete example of how to integrate our approach with development processes, we implemented a prototype tool that analyzes an existing repository to identify one-day cross-fork vulnerabilities in its dependencies. The tool currently supports two dependency mechanisms: Git submodules (common in C/C++) and `go.mod` files (used in Go projects). Using the results from our large-scale analysis, we built a database mapping SWHIDs to vulnerability IDs.

The database is used as knowledge base to assess submodules or Go dependencies. For submodules, the parent repository is cloned to extract the commit hashes referenced by each submodule. These hashes are then matched against the database to identify associated vulnerabilities. For Go, the `go.mod` file is parsed to extract dependencies that specify GitHub repositories and tags. The GitHub API is queried to resolve the tag to a specific commit, which is then checked against our knowledge base of known vulnerabilities.

### C. Evaluation

We evaluated our prototype tool by conducting one synthetic experiment and with a larger-scale empirical analysis in the real world. For the synthetic experiment, we created an example repository depending on panda-re via a Git submodule pointing to an old vulnerable commit (`f052389a634debd148e820d6bf88b5a77fe670d7`). Our tool automatically identified that the submodule is affected by CVE-2019-13164 (discussed previously), showing that this approach can identify vulnerabilities inherited via Git submodules.

Next, we crawled the 300 most popular Go repository containing a `go.mod` file, and used the tool on each of them. This analysis allowed us to find 10 unique potentially vulnerable dependencies, effectively used in 21 repositories out of the 300 most popular ones. This analysis provides a first feedback that could be analyzed further by the go community, enhancing the reliability and the confidence in their open-source ecosystem.

The tool currently runs using an unfiltered knowledge base, prior to the strict filtering of Figure 9, which might lead to false positives. It is just meant to be a proof-of-concept that constitutes a first step towards automating the detection of one-day vulnerabilities inherited via forks in real-world scenarios.

## VI. Discussion

### A. Implication on development practices

*a) Forking is easy, maintaining is hard:* With the advent of collaborative platforms such as GitHub, forking a repository is now possible with a single click. However, creating a hard fork is not as straightforward as it appears. Maintaining a fork is challenging and in practice translates to only "infrequent propagation" of changes [4] from upstream repository. Our analysis highlights a real issue concerning the maintenance of open-source forks, many vulnerabilities could be easily fixed, but are not because of the lack of a proper dedicated team associated with these projects.

Limiting vulnerability risk requires establishing a strategy for integrating upstream changes while limiting conflicts and manual intervention. These strategies, such as periodic *cherry-pick*, *merge*, or *merge rebase*, although they are used in practice[13], are only covered in a few studies from the literature [4]. Depending on the objective of forking and the level of divergence with the upstream repository, these strategies will ultimately require resources that some projects may not afford.

Many CVEs still fall under the radar of many of these forks. This underscores the need for automated tools, not only to notify maintainers but also to automatically apply fixes.

*b) Limiting false positive:* To limit false positives, fork maintainers can adopt strategies to apply security patches, such as using cherry-pick or merging when feasible and including the CVE ID in the patch commit if an alternative should be used. This method allows our approach to handle patches automatically and limit false positives. Additionally, our analysis can be restricted to vulnerabilities that affect specific folders in the upstream repository. For example, forks of the Linux kernel intended for ARM architecture are not impacted by vulnerabilities in x86-specific code.

*c) The opportunity of proper mapping between semantic versioning and version control:* In this work, we demonstrate the potential of using version control systems (VCS) to handle vulnerability analysis, focusing on propagating information about vulnerability introductions and fixes across the open-source ecosystem. This analysis has been possible thanks to a precise mapping between declared impacted software versions and its version history, here a commit hash. An important portion of the OSV reports declare only introductions and fixes using semantic versioning and were therefore excluded from our analysis. Thus, providing a proper mapping between the source code and semantic versioning represents an opportunity to increase the scope of the analysis and cover more CVE.

### B. Threats to validity

*a) Conclusion validity:* Our analysis is based on a representation of the global commit graph provided by Software Heritage. Therefore, our approach is only applicable to projects for which the commit history is publicly accessible, excluding closed-source forks. Nevertheless, our approach could be used for analyzing the closed-source commit graph of an organization's codebase.

Our approach tracks the introduction and fixes of vulnerabilities across Git history, with the detection of commits having the introduction commit in their ancestors but that lack the corresponding fix commit. However, as discussed in section VI, there are multiple reasons why a commit in such a situation is not affected by a vulnerability. To mitigate this threat, we use the term "potential" or "presumed" vulnerable.

*b) Reliability:* The results obtained in this work can be reproduced with the same input data. The manual analysis protocol is susceptible to human error, though we mitigated this risk with a rigorous protocol that involves the validation of the fork maintainer.

*c) Internal validity:* Our methodology (cf. fig. 9) relies on GitHub popularity metrics to identify forks of interest, i.e., forks with distinct user bases. Different filtering conditions can be designed based on different platform-specific popularity metrics or constraints based on intrinsic metadata [22], with the aim of detecting more forks of interest.

The strict filtering methodology aims to automatically exclude pairs of $\langle fork, vulnerability \rangle$ that are at high risk of being false positives. The objective was to demonstrate that our approach succeeded in identifying real one-day vulnerabilities within a limited number of manual evaluations (here only 65 pairs). However, this strict filtering might have dropped true positives, for instance, unpatched heads on non-default branches that still have users (e.g. long term support version).

*d) Construct validity:* The vulnerability ranges are augmented with cherry-picks of referenced events such as introduction or fix. However, the current approach only handles commits that explicitly mention "(cherry picked from

[13] https://web.archive.org/web/20250114052242/https://github.blog/developer-skills/github/friend-zone-strategies-friendly-fork-management/

commit ...)" in their commit message (cf -x option of git cherry-pick command) [14]. Detecting all the cherry-picks at the global level remains future work. To mitigate this threat, we performed a local analysis based on the git repository during strict filtering.

*e) External validity:* The representation of the global commit graph used is the one provided by Software Heritage. Therefore, our analysis is dependent on the Software Heritage archive and can be impacted by inconsistencies in the archived data. The quality of the vulnerability database also impacts our analysis. Incorrect "fixed" commits, which do not contain the actual fix, such as those filtered in section IV-B can prevent the evaluation of the fork impact. However, it does not necessarily mean that the propagation of such events is not correct, as they can indeed be assimilated to "last_affected".

## VII. Related work

*a) Cross-fork vulnerability propagation (at small scale):* Yi et al. [35] identified the propagation of 116 vulnerabilities in 16 forks of two blockchain projects, using code clone detection techniques. They then characterized the propagation of vulnerabilities and patching practices within the blockchain ecosystem. Their analysis enabled to discover unpatched vulnerabilities in 13 out of 16 forks. In contrast, our study is much larger scale (thousands of projects, millions of forks) and not restricted to forks of specific projects on specific platforms (we look at the global commit graph instead, using SWH as its proxy).

*b) Vulnerability propagation across dependencies:* The propagation of vulnerabilities through dependency graphs has garnered significant scholarly attention in recent years [37]. Recognized as a critical threat to the security of the open source software supply chain [9], [16], this phenomenon has been extensively investigated across major software ecosystems [36], [25], [30], [18], [21], [26]. In parallel, the emergence of tools like Renovate[15] and Dependabot[16] has spurred a growing body of research, highlighting both their potential benefits and inherent limitations in mitigating such risks [1], [23], [32], [14], [19]. Building upon this foundational work, this paper shifts the focus from dependency-based reuse to fork-based reuse. While forks are popular in open source, their role in the propagation of vulnerabilities remains unexplored. Our work addresses this gap, offering new insights into the security implications of forking.

*c) Vulnerable commit detection and dataset:* Zuo et Rhee [40] conducted a literature review on the usage of commit mining for discovering vulnerabilities. They find that multiple techniques are used to identify vulnerabilities, from "Hand crafted features based method" to automatic methods such as "Semantics learning based methods" or "Commit message study". More ad-hoc work focusing on dataset creation has been conducted, using methods such as automatic classification [27], static code analysis [11], existing Security Bulletins, Source Code and CVE Databases [5], or NVD database and manual analysis [12]. Our work is complementary to these, and modular in what is used as "ground truth" dataset for determining which commits introduce or fix vulnerabilities, to be propagated to forks.

*d) Vulnerability lifecycle analysis:* Some works looked into the patch development lifecycle through metrics like their "CVE lifespan", [17], [31], distribution [17] or dependencies and recurrences of vulnerabilities [20]. Tan et al. [34] studied the application of patches related to 806 CVEs, across 608 stable branches of 26 popular OSS projects. They observed that 80% of CVE-Branch pairs are unpatched; although considered stable, numerous branches are not maintained, resulting in the lack of patch propagation. Their observation, conducted at the granularity of repository branches, provides further evidence about the need of automated methods and tools to track vulnerability propagation across forks.

## VIII. Conclusion

In this paper, we leverage the global commit graph for tracking, at large scale, one-day vulnerabilities (i.e., known but unpatched vulnerabilities) across forks. Our analysis underscores the far-reaching impact of vulnerabilities in millions of open source projects. But more importantly, the opportunity for fork ecosystem to leverage their common source code and history to benefit security patches from each other, through automated tooling.

Specifically, by chasing one-day vulnerabilities associated with 7162 initial repositories, we identified 86.6 M presumed-vulnerable new commits in forks and 1.7 M potentially unpatched forks. We validated our approach by manually vetting unpatched forks, ultimately obtaining confirmation by upstream maintainers of 3 high-severity one-day vulnerabilities in actively used forks.

*Perspectives:* This work paves the way to semi-automated analyses that could benefit fork maintainers, by notifying them of potential one-day vulnerabilities in their projects, as well as users that have potentially vulnerable forks in their software supply chains.

As a future perspective, our work can also explore the use of code clone detection techniques from the literature to enhance the detection of equivalent commits that introduce or fix vulnerabilities. Finally, we plan to integrate this approach in large-scale public code archives like Software Heritage, making it a publicly-available resource for all open-source maintainers and users.

[14]https://git-scm.com/docs/git-cherry-pick
[15]https://docs.renovatebot.com/
[16]https://app.sourceclear.io

## References


[1] Mahmoud Alfadel, Diego Elias Costa, Emad Shihab, and Mouafak Mkhallalati. On the use of dependabot security pull requests. In *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021, Madrid, Spain, May 17-19, 2021*, pages 254–265. IEEE, 2021.

[2] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the FREENIX Track: 2005 USENIX Annual Technical Conference, April 10-15, 2005, Anaheim, CA, USA*, pages 41–46. USENIX, 2005.

[3] John Businge, Moses Openja, Sarah Nadi, and Thorsten Berger. Reuse and maintenance practices among divergent forks in three software ecosystems. *Empir. Softw. Eng.*, 27(2):54, 2022.

[4] John Businge, Moses Openja, Sarah Nadi, and Thorsten Berger. Reuse and maintenance practices among divergent forks in three software ecosystems. *Empirical Softw. Engg.*, 27(2), March 2022.

[5] Alexis Challande, Robin David, and Guénaël Renault. Building a commit-level dataset of real-world vulnerabilities. In Anupam Joshi, Maribel Fernández, and Rakesh M. Verma, editors, *CODASPY '22: Twelveth ACM Conference on Data and Application Security and Privacy, Baltimore, MD, USA, April 24 - 27, 2022*, pages 101–106. ACM, 2022.

[6] Roberto Di Cosmo and Stefano Zacchiroli. Software heritage: Why and how to preserve software source code. In Shoichiro Hara, Shigeo Sugimoto, and Makoto Goto, editors, *Proceedings of the 14th International Conference on Digital Preservation, iPRES 2017, Kyoto, Japan, September 25-29, 2017*, 2017.

[7] Laura A. Dabbish, H. Colleen Stuart, Jason Tsay, and James D. Herbsleb. Social coding in github: transparency and collaboration in an open software repository. In Steven E. Poltrock, Carla Simone, Jonathan Grudin, Gloria Mark, and John Riedl, editors, *CSCW '12 Computer Supported Cooperative Work, Seattle, WA, USA, February 11-15, 2012*, pages 1277–1286. ACM, 2012.

[8] Ying Dong, Wenbo Guo, Yueqi Chen, Xinyu Xing, Yuqing Zhang, and Gang Wang. Towards the detection of inconsistencies in public security vulnerability reports. In Nadia Heninger and Patrick Traynor, editors, *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, pages 869–885. USENIX Association, 2019.

[9] William Enck and Laurie A. Williams. Top five challenges in software supply chain security: Observations from 30 industry and government organizations. *IEEE Secur. Priv.*, 20(2):96–100, 2022.

[10] Douglas Everson, Long Cheng, and Zhenkai Zhang. Log4shell: Redefining the web attack surface. In *Proc. Workshop Meas., Attacks, Defenses Web (MADWeb)*, pages 1–8, 2022.

[11] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N. Nguyen. A C/C++ code vulnerability dataset with code changes and CVE summaries. In Sunghun Kim, Georgios Gousios, Sarah Nadi, and Joseph Hejderup, editors, *MSR '20: 17th International Conference on Mining Software Repositories, Seoul, Republic of Korea, 29-30 June, 2020*, pages 508–512. ACM, 2020.

[12] Antonios Gkortzis, Dimitris Mitropoulos, and Diomidis Spinellis. Vulinoss: a dataset of security vulnerabilities in open-source systems. In Andy Zaidman, Yasutaka Kamei, and Emily Hill, editors, *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*, pages 18–21. ACM, 2018.

[13] 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.

[14] 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.

[15] Manuel Hoffmann, Frank Nagle, and Yanuo Zhou. The value of open source software.

[16] 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.

[17] Frank Li and Vern Paxson. A large-scale empirical study of security patches. In Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 2201–2215. ACM, 2017.

[18] Qiang Li, Jinke Song, Dawei Tan, Haining Wang, and Jiqiang Liu. Pdgraph: A large-scale empirical study on project dependency of security vulnerabilities. In *51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2021, Taipei, Taiwan, June 21-24, 2021*, pages 161–173. IEEE, 2021.

[19] Ruyan Lin, Yulong Fu, Wei Yi, Jincheng Yang, Jin Cao, Zhiqiang Dong, Fei Xie, and Hui Li. Vulnerabilities and security patches detection in OSS: A survey. *ACM Comput. Surv.*, 57(1):23:1–23:37, 2025.

[20] Bingchang Liu, Guozhu Meng, Wei Zou, Qi Gong, Feng Li, Min Lin, Dandan Sun, Wei Huo, and Chao Zhang. A large-scale empirical study on vulnerability distribution within projects and the lessons learned. In Gregg Rothermel and Doo-Hwan Bae, editors, *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, pages 1547–1559. ACM, 2020.

[21] Chengwei Liu, Sen Chen, Lingling Fan, Bihuan Chen, Yang Liu, and Xin Peng. Demystifying the vulnerability propagation and its evolution via dependency trees in the NPM ecosystem. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 672–684. ACM, 2022.

[22] Petr Maj, Stefanie Muroya, Konrad Siek, Luca Di Grazia, and Jan Vitek. The fault in our stars: Designing reproducible large-scale code analysis experiments. In Jonathan Aldrich and Guido Salvaneschi, editors, *38th European Conference on Object-Oriented Programming, ECOOP 2024, September 16-20, 2024, Vienna, Austria*, volume 313 of *LIPIcs*, pages 27:1–27:23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024.

[23] Hamid Mohayeji, Andrei Agaronian, Eleni Constantinou, Nicola Zannone, and Alexander Serebrenik. Investigating the resolution of vulnerable dependencies with dependabot security updates. In *20th IEEE/ACM International Conference on Mining Software Repositories, MSR 2023, Melbourne, Australia, May 15-16, 2023*, pages 234–246. IEEE, 2023.

[24] Linus Nyman and Tommi Mikkonen. To fork or not to fork: Fork motivations in sourceforge projects. *Int. J. Open Source Softw. Process.*, 3(3):1–9, 2011.

[25] Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. Vulnerable open source dependencies: counting those that matter. In Markku Oivo, Daniel Méndez Fernández, and Audris Mockus, editors, *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2018, Oulu, Finland, October 11-12, 2018*, pages 42:1–42:10. ACM, 2018.

[26] Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. Vuln4real: A methodology for counting actually vulnerable dependencies. *IEEE Trans. Software Eng.*, 48(5):1592–1609, 2022.

[27] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, pages 426–437. ACM, 2015.

[28] Antoine Pietri, Guillaume Rousseau, and Stefano Zacchiroli. Forking without clicking: on how to identify software repository forks. In Sunghun Kim, Georgios Gousios, Sarah Nadi, and Joseph Hejderup, editors, *MSR '20: 17th International Conference on Mining Software Repositories, Seoul, Republic of Korea, 29-30 June, 2020*, pages 277–287. ACM, 2020.

[29] 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.

[30] Serena Elisa Ponta, Henrik Plate, and Antonino Sabetta. Detection, assessment and mitigation of vulnerabilities in open source dependencies. *Empir. Softw. Eng.*, 25(5):3175–3215, 2020.

[31] Piotr Przymus, Mikolaj Fejzer, Jakub Narebski, and Krzysztof Stencel. The secret life of cves. In *20th IEEE/ACM International Conference on Mining Software Repositories, MSR 2023, Melbourne, Australia, May 15-16, 2023*, pages 362–366. IEEE, 2023.

[32] Hocine Rebatchi, Tegawendé F. Bissyandé, and Naouel Moha. Depend-

abot and security pull requests: large empirical study. *Empir. Softw. Eng.*, 29(5):128, 2024.

[33] 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.

[34] Xin Tan, Yuan Zhang, Jiajun Cao, Kun Sun, Mi Zhang, and Min Yang. Understanding the practice of security patch management across multiple branches in OSS projects. In Frédérique Laforest, Raphaël Troncy, Elena Simperl, Deepak Agarwal, Aristides Gionis, Ivan Herman, and Lionel Médini, editors, *WWW '22: The ACM Web Conference 2022, Virtual Event, Lyon, France, April 25 - 29, 2022*, pages 767–777. ACM, 2022.

[35] Xiao Yi, Yuzhou Fang, Daoyuan Wu, and Lingxiao Jiang. Blockscope: Detecting and investigating propagated vulnerabilities in forked blockchain projects. In *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023*. The Internet Society, 2023.

[36] Ahmed Zerouali, Tom Mens, Alexandre Decan, and Coen De Roover. On the impact of security vulnerabilities in the npm and rubygems dependency networks. *Empir. Softw. Eng.*, 27(5):107, 2022.

[37] Fangyuan Zhang, Lingling Fan, Sen Chen, Miaoying Cai, Sihan Xu, and Lida Zhao. Does the vulnerability threaten our projects? automated vulnerable API detection for third-party libraries. *IEEE Trans. Software Eng.*, 50(11):2906–2920, 2024.

[38] Shurui Zhou, Bogdan Vasilescu, and Christian Kästner. What the fork: a study of inefficient and efficient forking practices in social coding. In Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo, editors, *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, pages 350–361. ACM, 2019.

[39] Shurui Zhou, Bogdan Vasilescu, and Christian Kästner. How has forking changed in the last 20 years?: a study of hard forks on github. In Gregg Rothermel and Doo-Hwan Bae, editors, *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, pages 445–456. ACM, 2020.

[40] Fei Zuo and Junghwan Rhee. Vulnerability discovery based on source code patch commit mining: a systematic literature review. *Int. J. Inf. Sec.*, 23(2):1513–1526, 2024.