

Did You Forkget It? Detecting One-Day Vulnerabilities in Open-source Forks With Global History Analysis

Romain Lefeuvre, Charly Reux, Stefano Zacchiroli, Olivier Barais, Benoit Combemale

▶ **To cite this version:**

Romain Lefeuvre, Charly Reux, Stefano Zacchiroli, Olivier Barais, Benoit Combemale. Did You Forkget It? Detecting One-Day Vulnerabilities in Open-source Forks With Global History Analysis. Scored 2026 - Conference on Software Supply Chain Offensive Research and Ecosystem Defenses, Oct 2026, Pragues, Czech Republic. ⟨hal-05349203v3⟩

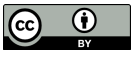
HAL Id: hal-05349203

https://hal.science/hal-05349203v3

Submitted on 14 Sep 2026

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

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



Distributed under a Creative Commons CC BY 4.0 - Attribution - International License

Did You Forkget It? Detecting One-Day Vulnerabilities in Open-Source Forks with Global History Analysis

Romain Lefeuvre
University of Rennes
Rennes, France
romain.lefeuvre@inria.fr

Charly Reux
University of Rennes
France, Rennes
charly.reux@inria.fr

Stefano Zacchiroli
LTCI, Télécom Paris
Institut Polytechnique de Paris
Palaiseau, France
stefano.zacchiroli@telecom-paris.fr

Olivier Barais
University of Rennes
France, Rennes
olivier.barais@irisa.fr

Benoit Combemale
Inria
France, Rennes
benoit.combemale@inria.fr

ABSTRACT

Tracking vulnerabilities inherited from third-party open-source software is a well-known challenge, often addressed by tracing the threads of dependency information. At scale, existing approaches precompute the vulnerable versions of software associated with known CVEs, based on declared impacted versions or using *local history analysis*. However, vulnerabilities can also propagate through *forking*: a repository forked after a vulnerability is introduced but before it is patched may remain vulnerable long after the original repository has been fixed. Existing *history analysis* approaches analyze only the repository referenced by the CVE, providing a local view of the ecosystem that excludes forks sharing part of its development history. Vulnerabilities disclosed and patched elsewhere in a fork ecosystem may therefore persist as one-day (known but unpatched) vulnerabilities without fork maintainers' awareness. This paper proposes a *global history analysis* approach that conforms to the evaluation semantics defined by the OpenSSF Open Source Vulnerability (OSV) format, while extending them from repository-local histories to the global commit graph of the open-source ecosystem. Leveraging the graph of public code captured by Software Heritage, our approach propagates vulnerability introduction and fix information across shared commit histories and performs automated impact analysis. Starting from 7162 repositories containing vulnerable commits listed in the OSV.dev vulnerability database, we propagate vulnerability information to 2.2 million forks. We evaluate our approach on a sample of 195 ⟨*fork*, *vulnerability*⟩ pairs from popular repositories, manually auditing their code and contacting their maintainers for confirmation and responsible disclosure. This process identified 135 high-severity one-day vulnerabilities, achieving a precision of 0.69, with 9 cases confirmed by maintainers.

KEYWORDS

one-day vulnerabilities, software supply chain security

1 INTRODUCTION

Open-Source Software (OSS) plays a critical role in the global digital infrastructure [19], its popularity having increased greatly over the past decades thanks to its unrivaled ease of code reuse. OSS reuse is now a central practice in the software supply chain but can be considered a "double-edged sword" [16], offering security benefits by involving a broader set of actors in maintaining code security, while also posing risks when vulnerabilities in reused software are not fixed in a timely manner or when reuse increases the attack surface (e.g., supply chain attacks).

There are two main mechanisms for the reuse of OSS [4]: *dependencies* and *forks*. The dependency mechanism involves calling into the code of a separate, freely licensed software via its API. Forking means reusing the source code of an existing OSS to establish a separate development trajectory, resulting in distinct software [50]. Forking serves multiple purposes: facilitating large-scale development through a GitFlow-like approach [38], addressing conflicts within the community, or extending functionality without integrating new ideas upstream [33, 51]. This can be observed empirically: more than 40% of repositories hosted on GitHub are forks[1].

The security of the global open-source ecosystem has attracted significant attention in recent years, particularly in the wake of high-profile software supply chain attacks on specific dependencies [24]. To address these growing risks, governments have introduced regulations requiring the software industry to identify and document the software supply chain. In the United States, Executive Order 14028 [44] pushes for the use of Software Bills of Materials (SBOMs), defined as "formal records containing the details and supply chain relationships of the various components used in building software." Similarly, the European Union's Cyber Resilience Act (CRA) [14] strengthens the security standards for digital products and holds "manufacturers" of digital services responsible for security issues in their supply chain, including those arising from reused software. These regulations place responsibility on service manufacturers to conduct appropriate audits and remediate vulnerabilities related to reused code.

Most research in this area has focused on the challenges associated with secure dependency management [18]. The risk of vulnerability propagation through dependencies is well documented in

To appear in Scored, 2026, Preprint
2026. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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

the literature, which emphasizes 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 need to assess dependency-related vulnerabilities has driven the development of tools such as GitHub Dependabot [15] and OSV-Scanner [17], which leverage vulnerability databases built from either manual annotations or *history analysis*.

The security implications of fork-based reuse have received comparatively less attention, in stark contrast to how widespread and foundational fork-based reuse is in major OSS: the Linux kernel, for instance, has numerous forks, including prominent derived kernels such as Android.

Problem statement. This paper focuses on the underexplored 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 initial ("upstream") code repository are incorporated into ("downstream") forks. For example, if a downstream code repository is forked after an upstream commit that introduces a "high severity" vulnerability and before the corresponding commit that fixes it (assuming it already exists), the downstream code repository will most likely remain vulnerable until a fixing commit is merged back there. From the moment the vulnerability is discovered and the fix commit is integrated, the downstream code repository is affected by a so-called *one-day vulnerability*: a known, but unpatched vulnerability.

Existing history analysis approaches, used to identify the vulnerable software versions impacted by a CVE, analyze only the repository referenced by the CVE, providing a local view of the ecosystem that excludes forks sharing part of its development history. Vulnerabilities disclosed and patched elsewhere in a fork ecosystem may therefore persist as one-day (known but unpatched) vulnerabilities.

Given the current lack of approaches and tools to track cross-fork vulnerability propagation, fork maintainers are tasked with manually tracking vulnerabilities associated with their fork ecosystem 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 be unaware 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 and fix security vulnerabilities; and (3) propagating the information about which commits are vulnerable, from upstream OSS 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…). To tackle this challenge, we propose shifting from repository-local history analysis to a global history analysis of the open-source ecosystem. Rather than analyzing each repository in isolation, our approach operates on a representation of the global commit graph that connects repositories through their shared development histories, enabling vulnerability information to be propagated across entire fork ecosystems.

Contributions. This paper introduces the following contributions:

- A *global history analysis* approach that conforms to the evaluation semantics defined by the OSV format, while extending them from repository-local histories to the global commit graph of the open-source ecosystem, thereby enabling fork maintainers to identify one-day vulnerabilities by tracking introducing and fixing commits across fork ecosystems.
- An implementation of the approach that can scale to the complete Software Heritage commit graph (consisting of more than 5 billion unique commits), enabling commit-level vulnerability tracking across heterogeneous public forges and version control systems.
- A large-scale study showing that, starting from 7162 repositories, referenced in the OSV database as having been affected by vulnerabilities in the past, vulnerabilities propagate to 2.2 million potentially impacted forks.
- The evaluation of the approach on a sample of real, high-impact one-day vulnerabilities in independent forks. After filtering for significant use (based on GitHub stars) and severity (based on CVSS scores), we identified 135 cases of vulnerabilities (precision 0.69) and obtained further positive confirmation from maintainers for 9 high-severity one-day vulnerabilities.

Data availability statement. A replication package of the experiments presented in this paper is available online [39].

2 BACKGROUND

Forking. This open-source practice consists in creating a new code repository from an existing one, usually preserving prior development history. Traditionally, "forking" meant splitting development efforts to bring an OSS in a different technical direction than intended by current maintainers, resulting in distinct software, referred to as a *hard fork*. With the advent of collaborative code hosting platforms [10], a new kind of fork has emerged, called *social fork*: contributors create personal forks of an OSS to propose changes via pull/merge requests [51]. Social forks tend to be ephemeral, but some grow their own user base, blurring the lines between hard and social forks.

In this paper, to *discover forks* of an OSS, we rely on the structural properties of the global commit graph produced by distributed version control systems (DVCSs) like Git, using the definition of *shared commit fork* from Pietri et al. [35]: "*a code repository B is a shared commit fork of code repository A, […] if there exists a commit c contained in the development histories of both A and B.*" This notion is undirected—*A* is a fork of *B and vice-versa* as soon as they share a commit—so we refer to a *fork ecosystem* as a group of repositories that are all forks of each other. This definition is broader than the forge-specific "fork" relation, and enables detecting forks that were not created through the "fork button", including repositories pushed independently and "cross-forge" forks hosted on different platforms (e.g., a GitLab.com fork of a GitHub repository).

Detection of vulnerabilities induced by code reuse. Analyzing dependencies at scale is now common practice with tools such as GitHub Dependabot [15] and OSV-Scanner [17]: they first build a

database mapping known vulnerabilities to affected software versions, then analyze a given repository's dependencies against that database. Ideally, static or dynamic detection techniques (cf. Section 5) would identify affected versions for the database, but these are often language-specific and too resource-intensive to apply at scale, as they typically require file-level analysis. Instead, many tools rely either on manually declared affected-version lists in CVE records, or on techniques that mine version control history to automatically identify vulnerable versions (e.g., OSV). We refer to the latter as *history analysis techniques*: unlike static or dynamic analysis, they rely solely on the commit graph, identifying versions that contain the vulnerability-introducing commit without the corresponding patch commit, which lets them scale efficiently by avoiding file-level analysis.

OSV (Open-Source Vulnerabilities). This OpenSSF project[2] consists of (1) a standard format[3] to describe vulnerabilities affecting OSS, and (2) a public database of vulnerability advisories in that format, aggregating security information from more than 20 software ecosystems. The OSV format associates vulnerabilities with *version ranges*, described as sequences of *events*. For example, an *introduced* event refers to the version where the vulnerability was introduced, and a *fixed* event refers to the version containing the patch. The format also defines events related to practical requirements, such as *last_affected*, which is used to reference the last known version with the vulnerability when the exact patch version is not known. Versions can be specified as version numbers or Git commit SHA1s; reasoning at the commit level enables more fine-grained analyses than version numbers and is particularly useful when studying forks. In the remainder of the paper, we only consider OSV vulnerabilities whose ranges are described using Git ranges.

The OSV database backs an API that answers queries like *"Is version X of software Y affected by any known vulnerability?"*, where "version" can likewise be a Git commit identifier. To answer such queries, OSV periodically crawls the repositories associated with a vulnerability and evaluates an algorithm (part of the OSV standard), maintaining an up-to-date *commit* $\rightarrow^*$ *vulnerability* relationship. Crucially, OSV only knows about commits in the upstream repository, and would produce false negatives when queried about commits only found in downstream forks.

Software Heritage (SWH). The largest public archive of software source code and associated development history [8], SWH archives code from more than 400 million OSS projects coming from development forges (GitHub, GitLab, etc.) and package manager repositories, comprising to date more than 20 billion files and 5 billion commits[4]. The SWH data model is a deduplicated Merkle directed acyclic graph (DAG) structure [32], with nodes representing source code artifacts, including (Git) commits and releases, each identified by a unique and persistent "SoftWare Hash IDentifier" (SWHID)[5] that is, for commits, mutually derivable with the corresponding Git identifier. The SWH graph, the best current approximation of the global commit graph, is formed by around 50 billion nodes and 0.5 trillion arcs, and can be loaded into main memory using a compressed representation [36] that allows scale-up analyses that would be prohibitively costly in other representations.

In the SWH graph, repositories are represented as *origins* associated with *snapshots*, which reference commits corresponding to branch heads. The *deduplication* property of the SWH graph makes it particularly useful for studying forks: forks hosted on different platforms, as well as those independently pushed to the same platform, become part of the same connected component of the graph, allowing forks to be studied globally, as previously done in [35].

3 TRACKING VULNERABLE COMMITS ACROSS FORK ECOSYSTEMS

We propose to track commits that introduce and fix known vulnerabilities at a global-scale, rather than on a per-repository basis. We postulate, and later verify experimentally, that this novel approach will let us identify forks of open-source projects affected by one-day vulnerabilities, unknown to their maintainers and other downstream stakeholders in the open-source software supply chain.

3.1 Challenge

Current *history analysis* 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 code repository associated in the past with at least one vulnerability is cloned and analyzed separately to detect vulnerable commits. A database of vulnerable commits is then used as a backend for an API that allows looking up vulnerabilities associated with a given commit. Forks of OSS share part of a common development history with their parent code repository, potentially inheriting unfixed vulnerabilities. Local *history analyses* (i.e., analyses performed on a local model, such as an individual Git repository, as opposed to a global model, such as the Software Heritage (SWH) archive), induce a "blind spot" problem that can result in one-day vulnerabilities being overlooked in forks.

Let us look at a real case of an unpatched fork involving a vulnerability with a high severity on the Common Vulnerability Scoring System (CVSS), identified as CVE-2019-13164,[6] impacting PANDA, a fork of QEMU. QEMU [3] 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 for over a decade, rarely integrating commits from the upstream code repository. Owing to their initially shared development history, PANDA inherits vulnerabilities present in QEMU at the time of the fork, including CVE-2019-13164.

Figure 1 illustrates the vulnerable commits in QEMU Git history, identified with a local analysis such as OSV.dev. In this situation, the QEMU Git repository, referenced in the OSV vulnerability report, is cloned and commits between the introduction and fix commit are listed and considered as vulnerable. This specific vulnerability is considered to be already present in the initial Git commit of QEMU (`e63c3dc`) and was eventually fixed in commit `03d7712`.

[2] https://osv.dev
[3] https://ossf.github.io/osv-schema
[4] https://www.softwareheritage.org
[5] https://swhid.org, ISO/IEC standard 18670:2025
[6] https://nvd.nist.gov/vuln/detail/CVE-2019-13164
[7] https://panda.re

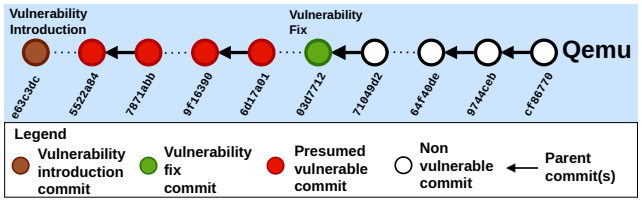


Figure 1: QEMU Git history: local repository-level analysis. Note that commits are built bottom-up on previous ones. Most recent commits are hence graph roots, shown on the right in the picture.

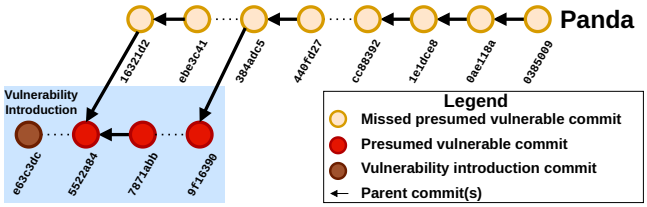


Figure 2: Propagation of vulnerable commits from upstream QEMU repository to its downstream PANDA fork.

Figure 2 depicts a portion of the PANDA Git history, which does not include the fix commit `03d7712`. The portion of the history highlighted in the blue box is shared with QEMU. The commits in this portion are identical to those in the QEMU repository, as indicated by their matching Git identifiers. Therefore, these commits are already considered vulnerable with QEMU local analysis (cf. Figure 1). 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 OSS is not referenced in the associated vulnerability report, leading to an overlooked one-day vulnerability. Note that the *new* vulnerable commits in PANDA history have commit identities (as SHA1 identifiers) missing from the upstream QEMU repository. Therefore they cannot be spotted as vulnerable based on commit identities only, because they will not be found in vulnerability databases which only crawl the original repositories or, at best, noteworthy hard forks.

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 code repository requires systematically examining open-source repositories across multiple code-hosting platforms. Even if a comprehensive list of public forks of a given vulnerable upstream code repository could be obtained, analyzing each fork individually would lead to significant redundancy in the effort. Forking a code repository is cheap, whereas vulnerability tracking is resource-intensive.

3.2 Approach

To address this challenge, we introduce a new *history analysis* approach based on a global model of commits, such as the one materialized by SWH (see Section 2). In this model, code histories are deduplicated and thus linked together through the identities of shared commits. Figure 3 recasts our QEMU/PANDA example

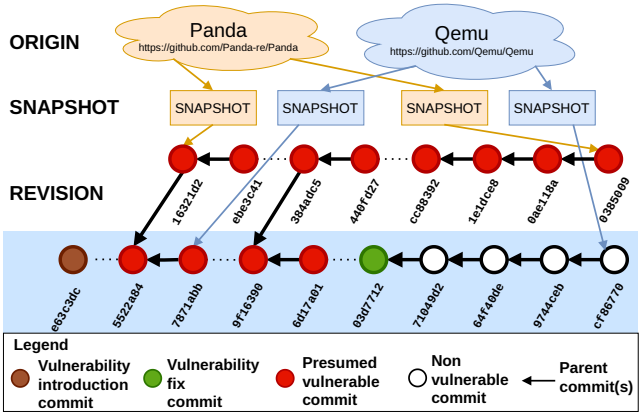


Figure 3: Running example on SWH model

in the context of a global commit model (here, SWH). The list of vulnerable commits can now be identified by listing commit nodes between introduction and fix commits. As a result, all 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.

To implement this idea in practice, we need to "label" each commit in the global commit graph with the set of known vulnerabilities affecting it. We obtain these labels by propagating commit-level information about which commits introduce or fix vulnerabilities from upstream repositories to downstream forks. Our algorithm formalizes the evaluation semantics of OSV,[8] which is applied in that context to individual repositories, and generalizes it to the global commit graph.

Given a vulnerability range R, we perform a depth-first traversal of the global commit graph starting from the introducing (`intro`) commits, propagating a `patched`/`unpatched` status forward to every reachable commit: a commit is marked patched as soon as it is itself a fixing (`fixed`) commit, or when any of its parents is already patched (so a merge commit is patched if at least one of its parents is). Beyond the `intro` and `fixed` events used here, the OSV standard also defines edge-case events—`last_affected` and `limit`—whose handling, together with reintroductions of the same vulnerability along different branches, follows the OSV evaluation semantics that our algorithm generalizes to the global commit graph. This process is repeated independently for each vulnerability range, and the results are aggregated into a global mapping from commits to the vulnerabilities affecting them. Although identifying vulnerable commits between introduction and fix may seem straightforward, this formalization is needed to correctly handle the full range of event types defined by the OSV standard. An implementation of our approach is available in the replication package [39].

We described our automated approach for propagating commit-level vulnerability information from a code repository history to its forks using a unified model at the global scale of all publicly

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

available commits. Before using this approach to look for actual one-day vulnerabilities (Section 4), we first need to check that it is practically feasible to propagate introduction/fix information across the global commit graph despite its size—from around 100K–1M commits for a single repository to billions of commits at the global scale. Once this is established, the propagation of introduction/fix information leads to the identification of *potentially* vulnerable commits in all forks of a given open-source project. Throughout this paper, *vulnerable* denotes the absence of a known patch, not demonstrated exploitability or code-path reachability, which we do not test. This label remains only *potential* at this stage, as forks that contain introducing, but not fixing, commits may still be unaffected in practice, for reasons we explore in Section 4.

3.3 Experimental protocol

To validate this propagation at scale, we execute a three-step experimental protocol to propagate OSV vulnerability information across the global commit graph from Software Heritage. Figure 4 presents an overview of the three steps of our experimental protocol.

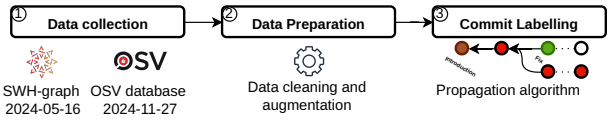


Figure 4: Overview of the experimental protocol steps for global propagation

Step 1: Data Collection. We retrieve two key datasets: (1) a recent export of the global commit graph from Software Heritage[9] (version 2024-05-16), and (2) vulnerability information exported from OSV[10] as of 2024-11-27.

Step 2: Data Preparation. We perform data cleaning and augmentation on the OSV dataset. We identify and exclude ranges with inconsistencies, such as multiple event types pointing to the same commit or formatting issues. We also exclude ranges containing events that reference commits absent from the SWH graph (e.g., commits more recent than our dataset snapshot). For data augmentation, we handle two special cases. First, when a vulnerability range specifies "0" as the introduction event (indicating the vulnerability existed from the repository's inception), we identify the leaf commits of the associated repositories to complete the range. Second, we detect cherry-picked events across branches (potentially spanning different forks) by identifying commit messages containing the default Git pattern "`cherry picked from commit...`". We add these detected cherry-picked commits as additional events within the same range.

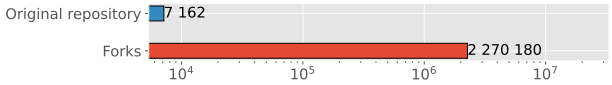
Step 3: Commit Labeling. We apply the propagation algorithm described in Section 3.2 to the SWH commit graph using the augmented OSV ranges. The output is a comprehensive mapping from all SWH commits to their associated OSV vulnerability ranges. All experiments were run on a server with 16 × 32 GiB DDR4 memory, 3200 MHz; 6 TiB of SSD storage; 2× AMD EPYC 7543 32-core CPUs, 2.8 GHz; Ubuntu 22.04.

[9]https://docs.softwareheritage.org/devel/swh-dataset/graph/dataset.html.
[10]https://google.github.io/osv.dev/data/#data-dumps

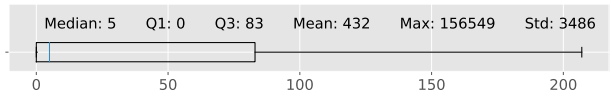
3.4 Results

Figure 5 shows that, after commit labeling (step 3), starting from a set of 7162 repositories referenced by OSV, we identified 2.2 million forks containing at least one presumed vulnerable commit. Note that we filter forks to keep only those containing at least one presumed vulnerable commit that is missing from the upstream repository. This excludes the trivial cases where downstream forks have not diverged from upstream in ways that are relevant for one-day vulnerabilities. Such cases can already be handled with state-of-the-art history analysis approaches, and are hence uninteresting for our experimental validation.

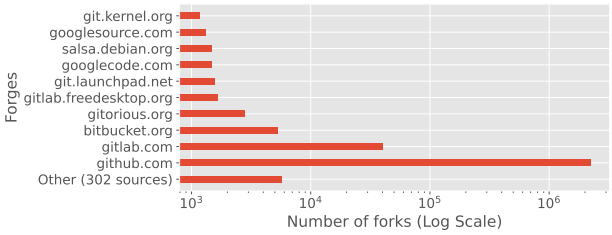
On average, each upstream repository affected at some point in its development history by at least one known vulnerability propagates it to 432 forks, which might remain affected. A first quartile of 0 and a median of 5 suggest that a few upstream repositories have a large number of potentially vulnerable forks. Although the majority of vulnerable forks are hosted on GitHub, potentially vulnerable commits are 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.



(a) Number of forks compared to upstream repositories



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



(c) Number of forks by forge

Figure 5: Forks of OSV-referenced repositories, having at least one new vulnerable commit.

As shown in Figure 6, we identified 158.9 million commits as potentially vulnerable, 86.3 million of which can only be found in fork histories (by commit identity), accounting for 54.3% of all potentially vulnerable commits.

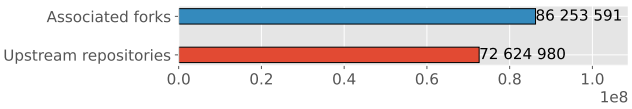


Figure 6: Number of new vulnerable commits in forks compared to upstream repositories

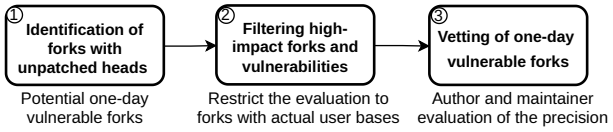


Figure 7: Overview of the experimental protocol steps

Takeaway: Starting from 7162 repositories, our *global history analysis* approach scales to the global commit graph, enabling tracking of vulnerability introductions and fixes across 2.2 million forks on 312 forges.

4 DETECTING ONE-DAY VULNERABLE FORKS

In the previous section, we propagated vulnerability information across the global commit graph. In this section, we evaluate our *global history analysis* in the scenario of detecting *one-day vulnerable forks*: forks that remain vulnerable in their latest revision to a vulnerability already patched elsewhere in their fork ecosystem.

Our *global history analysis* propagated vulnerability information across 2.2 million forks, identifying 86.3 million *potentially* vulnerable commits not present in upstream repositories. However, the presence of a vulnerable commit in the development history of a fork does not necessarily mean the fork is vulnerable in practice, for three reasons. First, the presumed vulnerable commit might not be the most recent (or "head") commit, to which users are most exposed. Second, forks may have applied a patch *equivalent* to the upstream fix through alternative means: they do fix the vulnerability, but are not recognized as doing so by our global history analysis. Third, forks might not actually be vulnerable if they have diverged significantly from the upstream codebase, making it impossible to trigger the original vulnerable code path.

To account for these limitations, we evaluate our approach on a sample of actively used forks affected by high-impact vulnerabilities. This sampling strategy serves two purposes: it maximizes the likelihood of receiving a response from maintainers, on which we rely to build our ground truth; and it evaluates our tool in the context it is intended for, namely helping maintainers of hard forks with real user bases who could be impacted by an undisclosed one-day vulnerability.

4.1 Research question

We proceed to investigate whether the *potential* vulnerabilities identified by our global propagation correspond to actual one-day vulnerabilities in real-world forked OSS. Thus, we introduce our research question: **RQ: Can we leverage the proposed *global history analysis* approach to detect at scale one-day vulnerabilities in real-world forked OSS?**

The next section presents the experimental protocol we used to answer this research question.

4.2 Experimental protocol

To answer this RQ, we propose a multi-stage experimental protocol composed of three phases: first, we identify forks with unpatched head commits that remain vulnerable; second, we apply a series of filters to focus on high-impact forks and vulnerabilities; third, we validate our findings through both manual code audit and confirmation from project maintainers. Figure 7 presents an overview of the three steps of our experimental protocol.

4.2.1 Identifying forks with unpatched heads. To identify one-day vulnerabilities, we focus on the most recent commit in a repository branch that remains vulnerable, which we call *unpatched head*. A fork containing an unpatched head is a *presumed one-day vulnerable fork*. This allows us to exclude forks that had vulnerable commits at some point in their history, but have since patched them. We further restrict this analysis to vulnerabilities specifying a `fixed` event: unlike `last_affected`, which only bounds the affected range without referencing the patch itself, a `fixed` event gives us a commit reference, which is necessary to evaluate whether the fix has since been incorporated into a given fork.

4.2.2 Filtering high-impact forks and vulnerabilities. We identified in the previous section a large number of forks impacted in their history by referenced vulnerabilities (cf. Figure 5a). However, many of these are likely inactive or social forks with no real user base [20, 51]. To answer this RQ, evaluating our approach for detecting real-world one-day vulnerabilities requires us to identify *high-impact forks*, which we define as forks with actual user bases and affected by high-severity vulnerabilities.

Popularity filtering. We first filter on the size of the user-base of a forked repository. To approximate this criterion, we rely on repository popularity metrics, specifically the number of stars and forks (of the fork itself, i.e., transitive forks of the upstream project) provided by GitHub. The filtering criteria were defined as follows: we only keep forks hosted on GitHub with more than 100 stars and more than 10 forks. Note that, while Software Heritage (SWH) is more general than just GitHub, from now on we restrict the analysis to only projects archived by SWH and originally hosted on GitHub. This is because SWH currently does not archive popularity metrics for platforms other than GitHub. Since the vast majority (97%, as shown in Figure 5c) of the forks detected as potentially vulnerable by our approach are hosted on GitHub anyway, this restriction has a limited quantitative impact.

Severity filtering. Second, we limit our analysis to vulnerabilities with CVSS severity of "high" or more (i.e., a CVSS base score ≥ 7.0). This allows us to focus on a limited number of potentially high-impact vulnerabilities, which we can in the following manually audit and contact maintainers about. (It would be unfeasible to do so for vulnerabilities of all severities identified thus far.)

Stale filtering. We exclude from further analysis forks that have been archived on GitHub or where the affected head commit is not

located in the main branch. The former denote products that are explicitly marked as no longer maintained; the latter are in general stale branches of limited impact on actual users. We also exclude repositories with no recent activity, i.e., a most recent commit with date older than 2023-01-01. Finally, we exclude cross-referenced pairs of ⟨*fork*, *vulnerability*⟩, i.e., forks that did not apply a specific vulnerability patch but have another vulnerability entry associated with a different patch.

Divergence filtering. Forks can diverge in ways that make a specific vulnerability not applicable to them. We use a heuristic to identify and exclude automatically divergent forks. We examine the files impacted by the fix commit. If at least one of the modified files is not present in the fork repository, we exclude the ⟨*fork*, *vulnerability*⟩ pair from further analysis. The rationale for this is that if the code that needs patching is no longer there, the code base is unlikely to be affected by the associated vulnerability. While executing this filtering step, we track file renames properly using `git log --follow --find-renames`.

4.2.3 Vetting of one-day vulnerable forks. Once we have filtered the forks and vulnerabilities of interest, we evaluate the extent to which our approach successfully identifies one-day vulnerable forks. We employ a two-phase validation procedure: In the first phase, we manually audit the source code for each ⟨*fork*, *vulnerability*⟩ pair, to identify potential false positives. In the second phase, we contact the corresponding fork maintainers to obtain further confirmation from them regarding the impact of the vulnerability. This also acts as responsibly disclosing the vulnerability to them, so that they can fix it if needed. In contrast to the general vulnerability detection literature, the detection of vulnerability propagation across fork ecosystems lacks a ground-truth dataset. We therefore treat maintainers' responses as the ground truth for evaluating the proportion of false positives produced by our approach, i.e., the maintainer-confirmation rate (the fraction of pairs flagged as vulnerable that are genuinely vulnerable, $TP/(TP+FP)$). However, in the absence of a comprehensive ground-truth dataset, we cannot reasonably estimate the proportion of false negatives (i.e., recall). Constructing a reliable ground-truth dataset for evaluating vulnerability propagation across fork ecosystems remains an open research problem.

Manual code audit. Before manually evaluating each ⟨*fork*, *vulnerability*⟩ pair, we conduct an evaluation of the data quality of the vulnerability report used as input of our approach. CVE databases are growing massively and rely heavily on unstructured data for which quality and consistency are a common concern [12]. Although it follows 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 precisely reference the actual vulnerability patches. Instead, fixed commits may point to catch-all release commits that include all changes associated with a new release, making it difficult to pinpoint the actual change that carried the security fix (cf. Section 6.3). Since these imprecise entries make it impossible to reliably assess whether a fork has incorporated the corresponding vulnerability patch, and thus impede the evaluation of our approach, we remove them from our analysis. In addition, the objective of this evaluation is to assess our approach, and not the quality of CVE reports.

Then, since the objective is to evaluate our approach rather than its current prototype implementation, we complement the global cherry-pick detection heuristic with a finer-grained one based on Git patch IDs.[11] While we already detected and propagated some cherry picks during the global propagation described in Section 3.3, we did so solely based on commit message traces, thereby missing cherry picks performed without the `-x` option. Importantly, patch ID-based detection is not conceptually tied to local analysis: it is a deterministic, per-commit computation that can technically be performed on the global commit graph, and its support over the entire Software Heritage archive is ongoing (cf. Section 6.3). It was, however, not yet available at the time of our experiments. Within the scope of the evaluation of our approach, we therefore apply the same detection locally, on a clone of the repository of each fork potentially affected by a vulnerability, and propagate the resulting cherry-pick information exactly as the global-scale implementation will. Since this correction is automatic and faithfully emulates the future global-scale behavior, evaluating our approach with it reflects the approach as deployed at scale; for transparency, we nevertheless also report the precision obtained without it.

Finally, an author evaluates all the remaining ⟨*fork*, *vulnerability*⟩ pairs to check whether an equivalent patch, not yet detected, had been applied in the fork, according to the following protocol:

- Read the CVE description.
- Read the diff of the commit that fixes the vulnerability.
- For each modified snippet ("diff hunk"), verify that it is indeed missing in the code base of the fork.
- If an alternative patch is found, mark the pair ⟨*fork*, *vulnerability*⟩ as false positive; mark it as true positive if not.

Maintainer notification. We contacted the maintainers of remaining forks identified as affected by 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 that exist: the maintainers. Based on the maintainer responses, we ultimately classify presumed vulnerable forks as:

- *False positive*: the fork is not vulnerable for reasons we could not identify on our own, 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.

4.3 Results

4.3.1 Forks with unpatched head. We identify 1.7 million presumed one-day vulnerable forks whose head commits remain unpatched. These account for 77.7% of all forks previously identified as containing at least one vulnerable commit in their histories (Figure 8), corresponding to 53 496 067 pairs of ⟨*fork*, *vulnerability*⟩.

4.3.2 Filtering forks and vulnerabilities of interest. Figure 9 presents the detailed filtering workflow and reports, for each macro step, the number of forks and vulnerabilities retained after filtering.

The popularity filtering (cf. Stage 1 of Figure 9) yields 1364 forks out of 1.76 million, confirming that only a small fraction of forks with a presumed one-day vulnerability are popular. The second

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

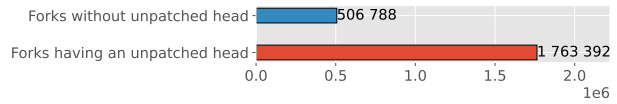


Figure 8: Forks having a presumed vulnerable commit in their history: proportion of forks having an unpatched head

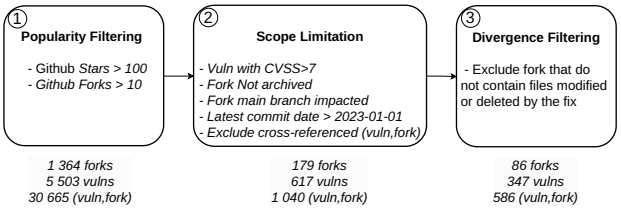


Figure 9: Result of the selection of forks and vulnerabilities of interest

step, limiting the scope of the analysis (cf. Stage 2 of Figure 9) yields 1040 pairs of ⟨*fork*, *vulnerability*⟩ out of 30 665. Detailed numbers of each sub-filter of this stage are available in the reproduction package [39]. The third step, filtering divergent forks (cf. Stage 3 of Figure 9), yields 586 pairs of ⟨*fork*, *vulnerability*⟩ with 347 distinct vulnerabilities and 86 forks of interest. The divergence heuristic filters out 43% of the pairs of ⟨*fork*, *vulnerability*⟩, meaning that the modified files associated with the vulnerability are not found in the fork. It confirms that, although sharing common history, forks can diverge significantly.

4.3.3 Vetting of one-day vulnerable forks. Now that we have selected forks and vulnerabilities of interest, we conducted the author and maintainer evaluation.

Author Evaluation. Figure 10 presents the execution of the author evaluation protocol, reporting for each step the number of forks and vulnerabilities retained after filtering.

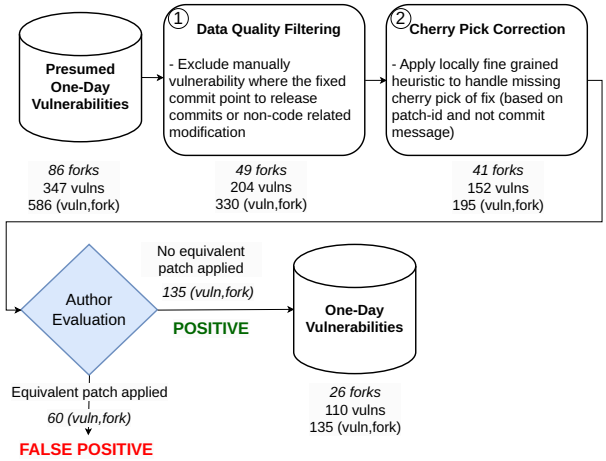


Figure 10: Author evaluation protocol

The data quality filtering (cf. Figure 10) yields 204 vulnerabilities out of the 347. We then applied the heuristic based on the patch ID of the fixing commits to identify and remove remaining cherry picks not yet handled at the global scale for implementation reasons (not a limitation of our approach). This heuristic yields 195 ⟨*fork*, *vulnerability*⟩ pairs associated with 152 unique vulnerabilities and 41 forks of interest. On average we observe 4.76 CVEs per fork with a median of 2, a first quartile of 1, a third quartile of 3 and a maximum of 100. Two outlier forks are present and associated respectively with 100 and 8 CVEs. All of these pairs are then manually evaluated to determine whether an equivalent patch has been applied or not (cf. Figure 10). This evaluation resulted in 135 positive and 60 false-positive ⟨*fork*, *vulnerability*⟩ pairs, corresponding to a precision of 0.69. This means that 69 % of evaluated pairs considered positive are actually positive based on author evaluation. Given that the two outlier forks account for more than half of all evaluated pairs, we conducted a secondary analysis in which these outliers were excluded. This second analysis yielded a precision of 0.6, showing the limited impact of outliers.

For transparency, we also report the precision obtained without the patch ID-based cherry-pick correction, i.e., as achieved by the current prototype implementation alone: it decreases to 0.41. This gap is entirely attributable to fixes cherry-picked without commit-message traces, which the prototype cannot yet recognize during global propagation. Since patch ID-based detection is technically applicable to the global commit graph and its integration into the Software Heritage graph is ongoing (cf. Section 6.3), we consider the corrected precision of 0.69 the relevant measure of the *approach*, while 0.41 characterizes the current *prototype* global propagation.

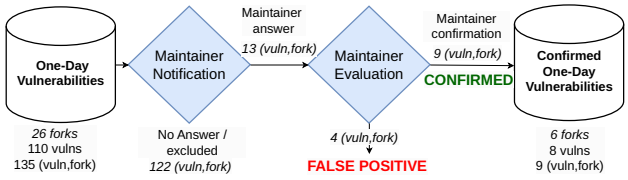


Figure 11: Maintainer evaluation protocol

4.3.4 Maintainer notification. Figure 11 presents the execution of the maintainer evaluation protocol: for each step, the number of forks and vulnerabilities retained after filtering. Out of the 135 ⟨*fork*, *vulnerability*⟩ pairs, we excluded 8 pairs because during the final due diligence before contacting maintainers, these forks were explicitly marked as deprecated or not intended for use (including some meant for educational purposes only). All the remaining fork maintainers have been notified. Among the candidates for confirmation, here are a few notable examples: `sonyxperiadev/kernel` (the community Linux kernel fork by Sony, for Xperia Android devices), `panda-re/panda` (our initial example), `reMarkable/linux` (the Linux kernel fork for reMarkable eInk tablets), and `go-nv/goenv` (a fork of pyenv, adapted to the Go ecosystem).

We received 13 responses, with 9 confirmed one-day vulnerabilities and 4 false positive ⟨*fork*, *vulnerability*⟩ pairs. For the remaining ones, we are awaiting confirmation. The confirmed one-day vulnerabilities result in a maintainer-confirmation rate of 0.69. This

means that 69 % of the pairs for which we received a maintainer response were confirmed. We received two types of responses: (1) from active communities where the maintainers of the repositories quickly responded 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 response is from `github.com/sonyxperiadev/kernel`, vulnerable to CVE-2021-45485 and CVE-2021-4154, whose maintainers fixed the vulnerability within days. The majority of the responses were of the second type, with maintainers acknowledging the vulnerable status of their fork, but lacking the resources to address it.

Our conclusion for this RQ is as follows:

RQ: *Global history analysis* is effective in supporting the identification of downstream forks affected by one-day vulnerabilities, as exemplified by the `PANDA` and `Xperia` cases. Among the 195 evaluated pairs of ⟨*fork*, *vulnerability*⟩, 135 remain positive after author evaluation (precision of 0.69). Maintainers confirmed 9 one-day vulnerabilities out of 13 responses (confirmation rate 0.69).

5 RELATED WORK

5.1 Detection of vulnerable dependencies

The propagation of vulnerabilities through dependency graphs has garnered significant scholarly attention in recent years [49], recognized as a critical threat to the security of the open-source software supply chain [13, 24] and extensively investigated across major software ecosystems [26, 34, 48]. In parallel, tools like Renovate[12] and GitHub Dependabot [15] have spurred a growing body of research on their potential and limitations [1, 18, 28]. 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 comparatively unexplored.

5.2 Detection of vulnerable fork versions

Yi et al. [47] identified the propagation of 116 vulnerabilities to 16 forks of two blockchain projects using code clone detection techniques, and found unpatched vulnerabilities in 13 out of 16 forks. In comparison, our study operates at a much larger scale (thousands of projects, millions of forks) and is not restricted to forks of specific projects or application contexts; instead, we analyze the global commit graph of public code, using SWH as its proxy.

In more exploratory contexts, Reid et al. [37], at the MSR 2021 Hackathon, used World of Code [30] to perform file history analysis: given a patch, they identify all previous versions of the modified files and consider them vulnerable, then identify forks whose HEAD contains such files. This file-based approach cannot handle projects that have modified the impacted files. In contrast, our approach is based on commit history analysis, allowing us to detect all commits that have the vulnerability-introducing commit in their ancestry without containing the corresponding fix.

[12] https://docs.renovatebot.com/

5.3 Detection of vulnerable software versions

A large body of work detects vulnerabilities within specific software versions, which Lin et al. [28] classify into *traditional* and *neural-network-based* detection. Traditional techniques include code clone detection at various granularities (text-, token-, tree-, graph-, and measure-based) [9, 23, 27, 41, 45], binary code analysis for closed-source software [6, 11], and formal methods such as symbolic execution [5], fuzzing [7], and taint analysis [21]. Neural-network-based approaches instead learn vulnerability patterns from curated datasets, using CNNs [40], GNNs [52], or NLP-based models [22].

5.4 Hybrid history analysis

Closer to our approach, some techniques combine vulnerability detection with Version Control System history analysis to avoid analyzing every version individually. Several build upon SZZ [42] and its variants, originally designed to identify bug-introducing commits from fixing commits. For instance, V-SZZ [2] uses static analysis to identify bug-inducing commits and then traverses the Git history to identify vulnerable commits associated with version tags. These algorithms address a fundamentally different problem from ours: they start from a commit known to fix a vulnerability and trace back the introducing commit, whereas we track forked repositories where, by definition of a one-day vulnerability, the fixing commit is not present yet. The challenges we address are therefore different: identifying all still-affected repositories (a scalability issue) and identifying impacted commits that may have different identifiers across forks (an identification issue), both of which we address by propagating OSV semantics across the global commit graph. Wu et al. [46] compare the execution time of six SZZ-derived approaches on 102 CVEs within 79 libraries and more than 12,073 library versions, showing that these take, on average, between 0.01 and 187 seconds per library version, and between 1.42 seconds and 17 771 seconds for analyzing all versions for a single vulnerability. In our setting, we analyze 53 million ⟨*fork*, *vulnerability*⟩ pairs across 1.7 million forks; applying prior approaches at this scale would require between 2.4 years and 30 thousand years under the same experimental conditions, and this estimate still assumes all forks are already identified and locally available—itself infeasible without a global model like ours. We instead reuse an existing database (OSV) of introduction and fix commits and rely solely on commit history analysis, which enables our approach to operate at this scale; we acknowledge, however, that OSV inherits known quality issues [12], e.g., conservative declarations of the introduction commit, which SZZ-based approaches could help refine at a smaller scale.

Finally, several studies analyze patch propagation and CVE lifespans at the repository or branch level [25, 29]. Notably, Tan et al. [43] found that 80% of CVE–branch pairs remain unpatched across 608 stable branches of 26 popular OSS projects, corroborating, at the branch level, the maintenance gap that our fork-level analysis also uncovers (Section 6).

6 DISCUSSION

6.1 Ethical considerations

The vulnerabilities we study are one-day vulnerabilities: they are already public and patched in at least one repository of the fork

ecosystem, but remain unpatched in the forks we identify. Our disclosure followed a responsible-disclosure process: for every fork we confirmed as affected, we privately notified its maintainers through the project's established contact channels, gave them the opportunity to remediate before any public release, and offered assistance in doing so (cf. Section 4.2.3). We contacted only project maintainers, and only about vulnerabilities in their own code; no end users were involved and no personal data was collected.

6.2 Implication for development practices

Forking is easy, maintaining is hard. With the advent of collaborative platforms such as GitHub, soft-forking a code repository now takes a single click; maintaining a hard fork is not as straightforward, and in practice translates to only "infrequent propagation" of changes [4] from upstream to downstream repositories. Our analysis highlights a real maintenance issue: many vulnerabilities could be easily fixed, but are not, due to a lack of human resources in the downstream, forked project. Limiting this risk requires a strategy for integrating upstream changes while limiting conflicts and manual intervention. Such strategies (periodic *cherry-pick*, *merge*, or *merge rebase*), although used in practice[13], are only covered in a few studies [4], and ultimately require maintenance resources that many OSS projects simply do not have—underscoring the need for automated tools, not only to notify maintainers, like those we introduce here, but also to automatically apply fixes, a relevant direction for future work.

Limiting false positives. Fork maintainers can limit false positives by adopting patch strategies our approach can recognize automatically, e.g., cherry-picks or merges, or by including the CVE ID in the patch commit when an alternative patch is used. Beyond the divergence heuristic we propose in Section 4.2 (based on the presence of files modified by a vulnerability patch in a fork), other heuristics could be designed, e.g., restricting the analysis to vulnerabilities impacting specific files: the maintainer of `sonyxperiadev/kernel` noted that, maintaining a fork of the Linux kernel for the ARM architecture, they are not impacted by x86-specific vulnerabilities even when that code is present in their fork.

6.3 Threats to validity

Conclusion validity. Our analysis is based on the global commit graph provided by Software Heritage, so our approach only applies to forks whose commit history is publicly accessible, excluding closed-source forks—though it could similarly be used on the closed-source commit graph of an organization's codebase.

Reliability. Our results 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 culminating in validation by project maintainers.

Internal validity. Our approach (cf. Figure 9) relies on GitHub popularity metrics to identify forks with relevant user bases potentially affected by high-severity vulnerabilities. Different filtering conditions could be designed based on other platform-specific popularity metrics or intrinsic metadata [31]. The divergence filter is deliberately conservative: it excludes a ⟨*fork*, *vulnerability*⟩ pair as soon as a single file modified by the fix is absent from the fork. This can drop genuinely vulnerable pairs (false exclusions), and, conversely, the presence of the modified files does not guarantee that the vulnerable code path is still reachable. This heuristic therefore trades recall for precision, consistent with our goal of surfacing high-confidence one-day vulnerabilities for disclosure.

Construct validity. As detailed in Section 4.2.3, the current prototype implementation of global propagation detects cherry-picked fixes only through commit-message traces ("`cherry picked from commit . . .`", cf `git cherry-pick`[14]). Consequently, the global counts reported in Figure 8 have an uncorrected precision of 0.41, as measured on our evaluation sample. The reported corrected evaluation additionally uses patch IDs to detect cherry-picked fixes without message traces. Although applied locally to the evaluation sample, this deterministic technique can be implemented globally and therefore estimates the precision of the complete approach. The confirmed vulnerabilities are unaffected because this correction preceded the author evaluation, and archive-wide patch-ID computation is currently being integrated into Software Heritage. In addition, following OSV semantics, a merge may be labeled patched even if conflict resolution discards the fix.

External validity. Our analysis depends on the Software Heritage archive and can be impacted by inconsistencies in the archived data, as well as by the quality of the vulnerability database [12]. Incorrect "fixed" commits, which do not contain the actual fix, such as those filtered in Section 4.2.3, can prevent the evaluation of the fork impact. However, this does not necessarily mean that the propagation of such events is incorrect, as they can effectively be treated as "last_affected" events.

7 CONCLUSION

In this paper, we leverage the global commit graph to track, at a large scale, one-day vulnerabilities (i.e., known but unfixed vulnerabilities) across forks of open-source projects. More importantly, our analysis highlights the opportunity to leverage the shared development history between upstream projects and their forks to develop automated tooling that can reinforce the security of the global OSS supply chain.

Experimentally, by tracking one-day vulnerabilities associated with 7162 initial repositories, we identified 86.3 million presumed-vulnerable new commits (missing from the original upstream repository) in forks and 1.7 million potentially unpatched forks. We validated our approach by manually vetting unpatched forks, ultimately obtaining confirmation by upstream maintainers of 9 high-severity one-day vulnerabilities in actively used forks.

This work paves the way to semi-automated analyses benefiting fork maintainers, users, and security auditors, by notifying them of potential one-day vulnerabilities in relevant code repositories. It also demonstrates the need to study vulnerabilities induced by code reuse practices beyond explicit dependencies, a pressing concern under the Cyber Resilience Act [14], which requires that products be placed on the market free of known exploitable vulnerabilities,

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

[14] https://git-scm.com/docs/git-cherry-pick

turning systematic one-day vulnerability analysis from good practice into a compliance obligation.

ACKNOWLEDGMENTS

This work was made possible by Software Heritage, the universal source code archive: https://www.softwareheritage.org. This work was supported by France Agence Nationale de la Recherche (ANR), program France 2030, reference ANR-22-PTCC-0001.

REFERENCES

[1] Mahmoud Alfadel, Diego Elias Costa, Emad Shihab, and Mouafak Mkhallalati. 2021. On the Use of Dependabot Security Pull Requests. In *MSR*. IEEE, 254–265.
[2] Lingfeng Bao, Xin Xia, Ahmed E Hassan, and Xiaohu Yang. 2022. V-SZZ: automatic identification of version ranges affected by CVE vulnerabilities. In *ICSE*. ACM, 2352–2364.
[3] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *USENIX, FREENIX Track*. USENIX, 41–46.
[4] John Businge, Moses Openja, Sarah Nadi, and Thorsten Berger. 2022. Reuse and maintenance practices among divergent forks in three software ecosystems. *Empirical Software Engineering* 27, 2 (2022), 54.
[5] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. 2008. EXE: Automatically Generating Inputs of Death. *ACM Trans. Inf. Syst. Secur.* 12, 2 (2008).
[6] Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan. 2016. BinGo: cross-architecture cross-OS binary search. In *FSE*. ACM, 678–689.
[7] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. 2020. Savior: Towards bug-driven hybrid testing. In *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 1580–1596.
[8] Roberto Di Cosmo and Stefano Zacchiroli. 2017. Software Heritage: Why and How to Preserve Software Source Code. In *iPRES*.
[9] Lei Cui, Zhiyu Hao, Yang Jiao, Haiqiang Fei, and Xiaochun Yun. 2021. VulDetector: Detecting Vulnerabilities Using Weighted Feature Graph Comparison. *IEEE Transactions on Information Forensics and Security* 16 (2021), 2004–2017.
[10] Laura A. Dabbish, H. Colleen Stuart, Jason Tsay, and James D. Herbsleb. 2012. Social coding in GitHub: transparency and collaboration in an open software repository. In *CSCW*. ACM, 1277–1286.
[11] Yaniv David, Nimrod Partush, and Eran Yahav. 2018. FirmUp: Precise Static Detection of Common Vulnerabilities in Firmware. In *ASPLOS*. 392–404.
[12] Ying Dong, Wenbo Guo, Yueqi Chen, Xinyu Xing, Yuqing Zhang, and Gang Wang. 2019. Towards the Detection of Inconsistencies in Public Security Vulnerability Reports. In *USENIX Security Symposium*. USENIX Association, 869–885.
[13] William Enck and Laurie A. Williams. 2022. Top Five Challenges in Software Supply Chain Security: Observations From 30 Industry and Government Organizations. *IEEE Security and Privacy* 20, 2 (2022), 96–100.
[14] European Union. 2024. Regulation (EU) 2024/2847 of the European Parliament and of the Council of 23 October 2024 on horizontal cybersecurity requirements for products with digital elements and amending Regulations (EU) No 168/2013 and (EU) 2019/1020 and Directive (EU) 2020/1828 (Cyber Resilience Act). Official Journal of the European Union. https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=OJ:L_202402847
[15] GitHub, Inc. 2025. GitHub Dependabot. https://github.com/dependabot.
[16] Antonios Gkortzis, Daniel Feitosa, and Diomidis Spinellis. 2019. A double-edged sword? Software reuse and potential security vulnerabilities. In *ICSR*. Springer.
[17] Google. 2025. OSV-Scanner. https://github.com/google/osv-scanner.
[18] Runzhi He, Hao He, Yuxia Zhang, and Minghui Zhou. 2023. Automating Dependency Updates in Practice: An Exploratory Study on GitHub Dependabot. *IEEE Transactions on Software Engineering* 49, 8 (2023), 4004–4022.
[19] Manuel Hoffmann, Frank Nagle, and Yanuo Zhou. 2024. The Value of Open Source Software.
[20] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. Germán, and Daniela E. Damian. 2016. An in-depth study of the promises and perils of mining GitHub. *Empirical Software Engineering* 21, 5 (2016), 2035–2071.
[21] Wooseok Kang, Byoungho Son, and Kihong Heo. 2022. Tracer: Signature-based static analysis for detecting recurring vulnerabilities. In *CCS*. 1695–1708.
[22] Soolin Kim, Jusop Choi, Muhammad Ejaz Ahmed, Surya Nepal, and Hyoungshick Kim. 2022. Vuldebert: A vulnerability detection system using bert. In *ISSREW*. IEEE, 69–74.
[23] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. 2017. VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery. In *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 595–614.
[24] Piergiorgio Ladisa, Henrik Plate, Matias Martinez, and Olivier Barais. 2023. SoK: Taxonomy of Attacks on Open-Source Software Supply Chains. In *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 1509–1526.
[25] Frank Li and Vern Paxson. 2017. A Large-Scale Empirical Study of Security Patches. In *CCS*. ACM, 2201–2215.
[26] Qiang Li, Jinke Song, Dawei Tan, Haining Wang, and Jiqiang Liu. 2021. PDGraph: A Large-Scale Empirical Study on Project Dependency of Security Vulnerabilities. In *DSN*. IEEE, 161–173.
[27] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Hanchao Qi, and Jie Hu. 2016. VulPecker: an automated vulnerability detection system based on code similarity analysis. In *ACSAC*. 201–213.
[28] Ruyan Lin, Yulong Fu, Wei Yi, Jincheng Yang, Jin Cao, Zhiqiang Dong, Fei Xie, and Hui Li. 2024. Vulnerabilities and Security Patches Detection in OSS: A Survey. *Comput. Surveys* 57, 1 (2024), 23:1–23:37.
[29] Bingchang Liu, Guozhu Meng, Wei Zou, Qi Gong, Feng Li, Min Lin, Dandan Sun, Wei Huo, and Chao Zhang. 2020. A large-scale empirical study on vulnerability distribution within projects and the lessons learned. In *ICSE*. ACM, 1547–1559.
[30] Yuxing Ma, Chris Bogart, Sadika Amreen, Russell Zaretzki, and Audris Mockus. 2019. World of code: an infrastructure for mining the universe of open source VCS data. In *MSR*. IEEE, 143–154.
[31] Petr Maj, Stefanie Muroya, Konrad Siek, Luca Di Grazia, and Jan Vitek. 2024. The Fault in Our Stars: Designing Reproducible Large-scale Code Analysis Experiments. In *ECOOP*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 27:1–27:23.
[32] Ralph C. Merkle. 1988. A Digital Signature Based on a Conventional Encryption Function. In *CRYPTO '87*. Springer Berlin Heidelberg, Berlin, Heidelberg.
[33] Linus Nyman and Tommi Mikkonen. 2011. To fork or not to fork: Fork motivations in SourceForge projects. In *IFIP International Conference on Open Source Systems*.
[34] Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. 2018. Vulnerable open source dependencies: counting those that matter. In *ESEM*. ACM, 42:1–42:10.
[35] Antoine Pietri, Guillaume Rousseau, and Stefano Zacchiroli. 2020. Forking Without Clicking: on How to Identify Software Repository Forks. In *MSR*. ACM.
[36] Antoine Pietri, Diomidis Spinellis, and Stefano Zacchiroli. 2019. The software heritage graph dataset: public software development under one roof. In *MSR*. IEEE, 138–142.
[37] David Reid, Kalvin Eng, Chris Bogart, and Adam Tutko. 2021. Tracing Vulnerable Code Lineage. In *MSR*. IEEE, 621–623.
[38] Julio César Cortés Ríos, Suzanne M. Embury, and Sukru Eraslan. 2022. A unifying framework for the systematic analysis of Git workflows. *Information and Software Technology* 145 (2022), 106811.
[39] Romain Lefeuvre, Charly Reux, Stefano Zacchiroli, Olivier Barais, and Benoit Combemale. 2026. Did You Forkget It? Detecting One-Day Vulnerabilities in Open-source Forks with Global History Analysis – Reproduction Package. swh:1:rev:041860955d7ff36e5905c5301d1b46f66e43227f, link to latest version.
[40] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. 2018. Automated vulnerability detection in source code using deep representation learning. In *ICMLA*. IEEE.
[41] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. 2016. SourcererCC: scaling code clone detection to big-code. In *ICSE*.
[42] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When do changes induce fixes? *ACM sigsoft software engineering notes* 30, 4 (2005), 1–5.
[43] Xin Tan, Yuan Zhang, Jiajun Cao, Kun Sun, Mi Zhang, and Min Yang. 2022. Understanding the Practice of Security Patch Management across Multiple Branches in OSS Projects. In *WWW*. ACM, 767–777.
[44] United States Executive Office of the President. 2021. Executive Order 14028: Improving the Nation's Cybersecurity. https://www.govinfo.gov/content/pkg/DCPD-202100401/pdf/DCPD-202100401.pdf.
[45] Huihui Wei and Ming Li. 2017. Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code. In *IJCAI*. 3034–3040.
[46] Susheng Wu, Ruisi Wang, Kaifeng Huang, Yiheng Cao, Wenyan Song, Zhuotong Zhou, Yiheng Huang, Bihuan Chen, and Xin Peng. 2024. Vision: Identifying affected library versions for open source software vulnerabilities. In *ASE*. ACM.
[47] Xiao Yi, Yuzhou Fang, Daoyuan Wu, and Lingxiao Jiang. 2023. BlockScope: Detecting and Investigating Propagated Vulnerabilities in Forked Blockchain Projects. In *NDSS*. The Internet Society.
[48] Ahmed Zerouali, Tom Mens, Alexandre Decan, and Coen De Roover. 2022. On the impact of security vulnerabilities in the npm and RubyGems dependency networks. *Empirical Software Engineering* 27, 5 (2022), 107.
[49] Fangyuan Zhang, Lingling Fan, Sen Chen, Miaoying Cai, Sihan Xu, and Lida Zhao. 2024. Does the Vulnerability Threaten Our Projects? Automated Vulnerable API Detection for Third-Party Libraries. *IEEE Transactions on Software Engineering* 50, 11 (2024), 2906–2920.
[50] Shurui Zhou, Bogdan Vasilescu, and Christian Kästner. 2019. What the fork: a study of inefficient and efficient forking practices in social coding. In *ESEC/FSE*.
[51] Shurui Zhou, Bogdan Vasilescu, and Christian Kästner. 2020. How has forking changed in the last 20 years?: a study of hard forks on GitHub. In *ICSE*. ACM.
[52] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *NeurIPS* 32 (2019).