A Large-Scale Empirical Study of Security Patches

Frank Li Vern Paxson
{frankli, vern}@cs.berkeley.edu
University of California, Berkeley and International Computer Science Institute

ABSTRACT

Given how the "patching treadmill" plays a central role for enabling sites to counter emergent security concerns, it behooves the security community to understand the patch development process and characteristics of the resulting fixes. Illumination of the nature of security patch development can inform us of shortcomings in existing remediation processes and provide insights for improving current practices. In this work we conduct a large-scale empirical study of security patches, investigating more than 4,000 bug fixes for over 3,000 vulnerabilities that affected a diverse set of 682 open-source software projects. For our analysis we draw upon the National Vulnerability Database, information scraped from relevant external references, affected software repositories, and their associated security fixes. Leveraging this diverse set of information, we conduct an analysis of various aspects of the patch development life cycle, including investigation into the duration of impact a vulnerability has on a code base, the timeliness of patch development, and the degree to which developers produce safe and reliable fixes. We then characterize the nature of security fixes in comparison to other non-security bug fixes, exploring the complexity of different types of patches and their impact on code bases.

Among our findings we identify that: security patches have a lower footprint in code bases than non-security bug patches; a third of all security issues were introduced more than 3 years prior to remediation; attackers who monitor open-source repositories can often get a jump of weeks to months on targeting not-yet-patched systems prior to any public disclosure and patch distribution; nearly 5% of security fixes negatively impacted the associated software; and 7% failed to completely remedy the security hole they targeted.

1 INTRODUCTION

Miscreants seeking to exploit computer systems incessantly discover and weaponize new security vulnerabilities. As malicious attacks become increasingly advanced, system administrators continue to rely on many of the same processes as practiced for decades to update their software against the latest threats. Given the central role that the "patching treadmill" plays in countering emergent security concerns, it behooves the security community to understand the patch development process and the characteristics of the resulting fixes. Illuminating the nature of security patch development can inform us of shortcomings in existing remediation processes and provide insights for improving current practices.

Seeking such understanding has motivated several studies exploring various aspects of vulnerability and patching life cycles. Some have analyzed public documentation about vulnerabilities, such as security advisories, to shed light on the vulnerability disclosure process [14, 32]. These studies, however, did not include analyses of the corresponding code bases and the patch development process itself. Others have tracked the development of specific projects to better understand patching dynamics [18, 28, 41]. While providing insights on the responsiveness of particular projects to security issues, these investigations have been limited to a smaller scale across a few (often one) projects.

Beyond the patch development life cycle, the characteristics of security fixes themselves are of particular interest, given their importance in securing software and the time sensitivity of their development. The software engineering community has studied bug fixes in general [29, 33, 34, 42]. However, there has been little investigation into how fixes vary across different classes of issues. For example, one might expect that patches for performance issues qualitatively differ from those remediating vulnerabilities. Indeed, Zama et al.'s case study on Mozilla Firefox bugs revealed that developers address different classes of bugs differently [41].

In this work, we conduct a large-scale empirical study of security patches, investigating 4,000+ bug fixes for 3,000+ vulnerabilities that affected a diverse set of 682 open-source software projects. We build our analysis on a dataset that merges vulnerability entries from the National Vulnerability Database [37], information scraped from relevant external references, affected software repositories, and their associated security fixes. Tying together these disparate data sources allows us to perform a deep analysis of the patch development life cycle, including investigation of the code base life span of vulnerabilities, the timeliness of security fixes, and the degree to which developers can produce safe and reliable security patches. We also extensively characterize the security fixes themselves in comparison to other non-security bug patches, exploring the complexity of different types of patches and their impact on code bases.

Among our findings we identify that: security patches have less impact on code bases and result in more localized changes than non-security bug patches; security issues reside in code bases for years, with a third introduced more than 3 years prior to remediation; security fixes are poorly timed with public disclosures, allowing attackers who monitor open-source repositories to get a jump of weeks to months on targeting not-yet-patched systems prior to any public disclosure and patch distribution; nearly 5% of security

CCS '17, October 30-November 3, 2017, Dallas, TX, USA

ACM ISBN 978-1-4503-4946-8/17/10…$15.00
https://doi.org/10.1145/3133956.3134072

fixes negatively impacted the associated software; and 7% failed to completely remedy the security hole they targeted. The findings of our analysis provide us with insights that suggest paths forward for the security community to improve vulnerability management.

Our work provides several contributions. First, we specifically focus on security vulnerabilities and their patches. While aspects of our work have similarities to prior efforts from the software engineering community that examined general bug fixes [29, 33, 34, 42], we tease apart the differences between security fixes vs. other bug fixes. Second, we develop a large-scale reproducible data collection methodology and associated analysis that ties extensive meta-data on vulnerabilities and their patches with the software source codes and change histories. As best we know, such a diverse set of data has not been previously collected and used to explore security patch development at scale. Conducting such an analysis at scale provides a third contribution: some prior works have considered analyses somewhat similar, but restricted to a small handful of software projects (often only one). We develop robust metrics that one can compute across a diverse group of projects, supporting a range of generalizable results.

2 RELATED WORK

There has been a body of work that investigated aspects of the vulnerability life cycle. Frei et al. [14] and Shahzad et al. [32] conducted similar analyses based on public documentation from vulnerability databases and security advisories. For example, they compared a vulnerability's public disclosure date with announcement dates for fixed releases available for distribution, finding them concurrent in 60% of cases. Ozment et al. [28] investigated the evolution of vulnerabilities in the OpenBSD operating system over time, observing that it took on average 2.6 years for a release version to remedy half of the known vulnerabilities. Huang et. al. [18] manually analyzed 131 cherry-picked security patches from five open-source projects, demonstrating that there exist cases where patch development was lengthy and error-prone. Nappa et al. [27] shed light on the patch deployment process from an end-user perspective, analyzing when security updates were available to clients and how quickly clients patched. In our work, we extensively explore new aspects of patch development dynamics that require merging information collected from vulnerability databases with that gleamed from software source code, such as vulnerability life spans in the code base and the timeliness of patching the code base relative to public disclosure. In addition, we aim to generate generalizable insights by studying a diverse set of over 650 open-source projects.

Explorations of bug fixes in general (beyond just security bugs) have been performed in the software engineering community. Zhong and Su [42] conducted an empirical study of over 9,000 bug fixes across six Java projects. They framed their investigation around patch properties that would make them suitable for generation by automatic program repair, finding that the majority are too complex or too delocalized to likely be automatically created. Similarly, Park et al. [29] studied supplementary bug fixes, additional fixes produced when the initial fix was incomplete. Their analysis covered three open-source projects and showed that over a quarter of remedies required multiple patches. Sliwerski et. al. [33] investigated two projects and correlated updates that required fixes with the update sizes, finding larger updates were more likely to require subsequent fixes. Soto et. al. [34] applied common bug fix patterns to Java patches, finding that less than 15% could be matched.

While these works are similar in their focus on patch characteristics, they mostly were conducted at a smaller scale, and do not differentiate between different kinds of bugs. Security patches are of special interest, given their importance in protecting users and the time sensitivity of their development. We seek to tease apart the differences between security and non-security bug fixes, a distinction that has not been previously scrutinized extensively. Most relevant is a case study performed by Zama et al. [41] on security and performance fixes in Mozilla Firefox. They noted differing remediation rates and complexities between security and performance patches. Perl et. al. [30] also analyzed Git commits that fixed vulnerabilities to produce a code analysis tool that assists in finding dangerous code commits. They found that indicative features of problematic commits include code which handles errors or manages memory, or is contributed by a new project developer. Most recently, Xu et. al. [39] developed a method for identifying security patches at the binary level based on execution traces, providing a method for obtaining and studying security patches on binaries and closed-source software. These early findings highlight the importance of considering different types of software bugs; a deep understanding of security patches and their development process can inform the security community in matters related to vulnerability management.

3 DATA COLLECTION METHODOLOGY

To explore vulnerabilities and their fixes, we must collect security patches and information pertaining to them and the remedied security issues. Given this goal, we restricted our investigation to open-source software for which we could access source code repositories and associated meta-data. Our data collection centered around the National Vulnerability Database (NVD) [37], a database provided by the U.S. National Institute of Standards and Technology (NIST) with information pertaining to publicly disclosed software vulnerabilities. These vulnerabilities are identified by CVE (Common Vulnerabilities and Exposures) IDs [23]. We mined the NVD and crawled external references to extract relevant information, including the affected software repositories, associated security patches, public disclosure dates, and vulnerability classifications. Figure 1 depicts an overview of this process. In the remainder of this section, we describe these various data sources and our collection methodology.

Note that throughout our methodology, we frequently manually inspected random samples of populations to confirm that the population distributions accorded with our assumptions or expectations. We chose sample sizes (typically of 100) such that they proved manageable for manual analysis while large enough to reflect fine-grained aspects of population distributions.

3.1 Finding Public Vulnerabilities

We relied on the NVD to find publicly disclosed vulnerabilities. The NVD contains entries for each publicly released vulnerability assigned a CVE identifier. When security researchers or vendors

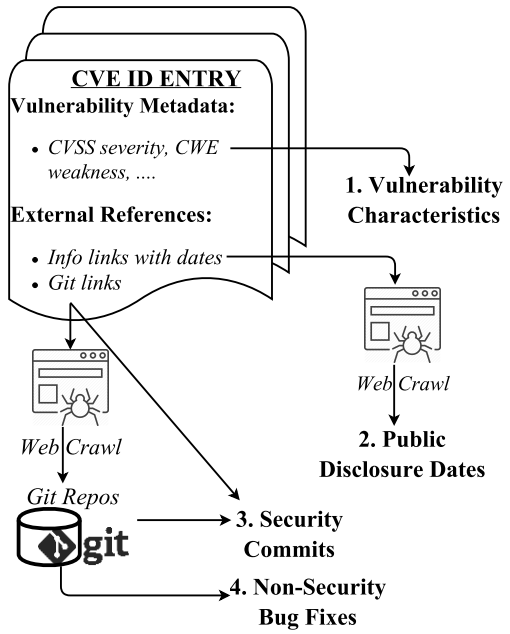


Figure 1: An overview of our data collection methodology. 1. We extracted vulnerability characteristics from CVE entries in the NVD with external references to Git commit links. 2. We crawled other references and extracted page publication dates to estimate public disclosure dates. 3. We crawled the Git commit links to identify and clone the corresponding Git source code repositories, and collected security fixes using the commit hashes in the links. 4. We also used the Git repositories to select non-security bug fixes.

identify a vulnerability, they can request a CVE Numbering Authority (such as the MITRE Corporation) to assign a CVE ID to it. At this point, information about the vulnerability may not yet be disclosed. Upon public release of the vulnerability information, the CVE ID along with its associated vulnerability information gets added to the CVE list, which feeds the NVD. NVD analysts investigate the vulnerability further, populating an entry for the CVE ID with additional information. In particular, they summarize the vulnerability, link to relevant external references (such as security advisories and reports), enumerate the affected software, identify the class of security weakness under the Common Weakness Enumeration (CWE) classifications [24], and evaluate the vulnerability severity using the Common Vulnerability Scoring System (CVSS) [12, 35].

While there exist other vulnerability databases (e.g., *securityfocus.com*, IBM's X-Force Threat Intelligence, and *securitytracker.com*), we focused on the NVD as it is: (1) public, free, and easily accessible in XML format, allowing for reproducibility and follow-on studies, (2) expansive, as the NVD aims to catalog all publicly disclosed vulnerabilities across numerous software packages, (3) manually vetted and curated, which in theory provides more accurate data, and (4) detailed, containing extensive documentation of vulnerabilities (notably external references).

We utilized the NVD XML dataset [38] as snapshotted on December 25th, 2016. Its 80,741 CVE vulnerabilities served as our starting point for further data collection.

3.2 Identifying Software Repositories and Security Patches

Many open-source version-controlled software repositories provide web interfaces to navigate project development (such as *git.kernel.org*). We frequently observed URLs to these web interfaces among the external references for CVE entries, linking to particular repository commits that fixed the security vulnerability. These links afforded us the ability to collect security patches and access the source code repositories.

As Git is arguably the most popular version control system for open-source software [31], we focused on references to Git web interfaces. This popularity was consistent with the CVE external references as well, where links to Git web interfaces were by far the most common. We observed more than 5,700 unique URLs with "git" as a substring, excluding those with another common substring "digit". To determine if these URLs were indeed related to Git, we randomly sampled 100 URLs. The vast majority of these were associated with Git web interfaces; only two out of the 100 URLs were non-Git URLs. In comparison, 1,144 external references contained "svn" (for SVN), 613 contained "cvs" (for CVS), and 347 contained "hg" or "mercurial" (for Mercurial), significantly fewer for these other popular version control systems compared to Git.

To find Git repositories and their security patches, we first reverse-engineered the URL paths and parameters used by popular Git web interfaces. These included cgit [2], GitWeb [6], github.com, and GitLab [5], and accounted for 95% of references with "git" as a substring. (Thus, to consider more Git web interfaces would have required additional URL reverse-engineering while producing diminished returns.) We also identified only an additional 128 URLs without "git" that were consistent with a common Git web interface, suggesting that we identified the majority of Git URLs. For the 80% of these Git URLs that linked to a particular commit (specified in Git by a commit hash), we crawled the web interfaces' summary/home pages and extracted the Git clone URLs, if listed.

In total, we retrieved 4,080 commits across 682 unique Git repositories, tied to 3,094 CVEs. Note that these repositories are distinct, as we de-duplicated mirrored versions. It is possible that some commits are not security fixes, as they may instead reference the change that introduced the vulnerability, or may contain a proof-of-concept exploit instead. However, we found that this is rarely the case. By manually investigating 100 randomly sampled commits, we found that all commits reflect fixes for the corresponding vulnerabilities.

3.3 Identifying Non-Security Bug Fixes

We can gain insight into any particularly distinct characteristics of security patches by comparing them to *non-security* bug fixes. However, to do so at scale we must automatically identify non-security bug fixes. We tackled this problem using a logistic regression that models the character n-grams in Git commit messages to identify likely bug fix commits.[1]

To train our commit classifier, we manually labeled 400 randomly selected commits drawn from all Git repositories as bug fixes or

[1]We also explored other commit features for classification, such as the number of files and lines affected by a commit, the type of commit changes (addition, deletion, modification), the day of week the commit was made, and the time since the previous commit. However, these did not provide adequate discriminating power.

non-bug fix commits (136 were bug fixes). We then featurized a commit message into a binary vector indicating the presence of common character n-grams in the commit message. To determine the size of n-grams, the threshold on the number of n-grams to include, and model parameters, we ran a grid search using 10-fold cross-validation on the training data. Our feature vector search space considered n-grams of lengths 2 to 10 and feature vectors that included the top 10,000 to the top 250,000 most frequently occurring n-grams for each class. Our model parameter search space considered both L_1 and L_2 regularization, with regularization strengths ranging from 0.1 to 10, and the inclusion of a bias term.

Our final classifier utilized n-grams of lengths 3 to 9, with feature vectors corresponding to the top 50,000 most common n-grams for each class. The model used L_2 regularization with a regularization strength of 1, and included a bias term. During 10-fold cross-validation, the classifier had an average recall of 82% and precision of 91%. While the classifier is not extremely accurate, it results in only a small fraction of false positives and negatives, which should have limited effect on the overall distributions of patch characteristics. In Section 5.2, we compare characteristics of security patches versus generic bug fixes. We manually validated that for these characteristics, the distribution of values is similar between our manually labeled bug fixes and our classifier-collected bug patches, indicating that our results for classifier-labeled bug fixes should be representative of randomly selected true bug fixes.

With our classifier, we collected a dataset of bug fixes by randomly selecting per repository up to 10 commits classified as bug fixes. (Fewer for repositories with less than 10 total commits.) We chose to select 10 commits per repository as that provided us with a large set of over 6,000 bug fixes (similar to our number of security fixes) balanced across repositories. Note that in our classifier training, security fixes were labeled as bug fixes. However, only 6% of bug fixes in our training data (a random sample) were security-related, thus our dataset consists almost entirely of non-security bug fixes.[2]

3.4 Processing Commits

For each commit we collected (both security and non-security patches), we extracted the historical versions of affected files both before and after the commit. The diff between these file versions is the patch itself. In addition, it is often useful to consider only changes to functional source code, rather than documentation files or source code comments. We processed the commit data using a best-effort approach (as follows) to filter non-source code files and remove comments, providing an alternative "cleaned" commit to analyze.

To do so, we mapped the top 30 most frequently occurring file extensions to the programming or templating languages associated with them, if any (e.g., an extension of *.java* corresponds to Java, whereas we assume *.txt* reflects non-source code). These included

[2] We also investigated developing a commit message classifier to automatically distinguish between security and non-security fixes, using as ground truth the manually-labeled commits as well as randomly selected CVE-related security fixes. Given the base rate challenge arising due to the relative rarity of security fixes, we found that the classifiers we tried did not provide nearly enough accuracy. We did not consider using patch characteristics (such as those explored in Section 5.2) as features as we aimed to understand how security and non-security bug fixes differed along these properties, thus using such features would provide skewed populations.

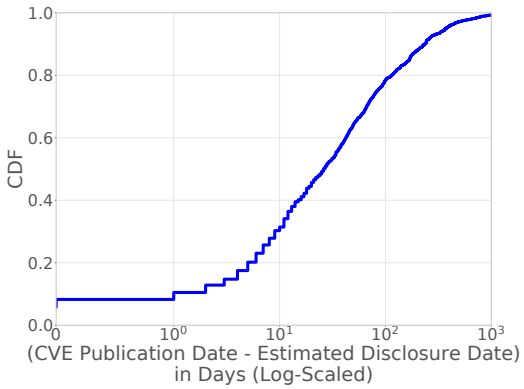


Figure 2: CDF of the number of days the estimated disclosure date precedes the CVE publication date.

C/C++, PHP, Ruby, Python, SQL, HTML, Javascript, Java, and Perl. We stripped comments and trailing whitespaces under the assumed programming language's syntax for source code files, and filtered out all other files. This provided a cleaned snapshot of files involved in a commit, from which we computed a cleaned diff.

This method is ultimately best-effort,[3] as we handled only the top 30 extensions and relied on extensions as file type indicators. However, we note that these top 30 extensions accounted for 95% of commit files, and incorporating additional extensions would have resulted in diminishing returns given that each extension potentially required a new cleaning process. Also, in a random sample of 100 files with a top 30 extension, all extensions corresponded correctly to the expected file type. This is unsurprising given these projects are open-source and often involve a number of developers, which likely discourages a practice of using non-intuitive and non-standard file extensions.

3.5 Estimating Vulnerability Public Disclosure Dates

Determining the public disclosure date of a vulnerability is vital to understanding the timeline of a vulnerability's life cycle. NVD entries contain a CVE publication date that corresponds to when the vulnerability was published in the database, not necessarily when it was actually publicly disclosed [36]. To obtain a more accurate estimate of the public disclosure date, we analyzed the external references associated with CVEs. These web pages frequently contain publication dates for information pertaining to vulnerabilities, which can serve as closer estimates of the public disclosure dates.

For the CVEs corresponding to our collected security commits, we identified the top 20 most commonly referenced sites that may contain publication dates, listed in Table 5 in Appendix A. Of these, two sites were no longer active (*mandriva.com* and *vupen.com*), one did not provide fine-grained dates (*oracle.com*), and IBM's Threat Intelligence X-Force site employed aggressive anti-crawling measures. For the remaining 16 sites, we constructed per-site parsers that extracted the date of the relevant publication for a given page. These pages include security advisories (such as from Debian and

[3] We also evaluated using the Linux "`file`" utility, but found it suffered from frequent errors.

Redhat), mailing list archives (e.g., *marc.info*, *openwall.com/lists*), other vulnerability database entries (e.g., *securityfocus.com*, *securitytracker.com*), and bug reports (such as Bugzilla bug tracking sites). We restricted our crawling to the top 20 sites, as each site required developing a new site parser, and we observed diminishing returns as we added more sites.

We crawled about 13,600 active external references in total, extracting a publication date from 94% of pages. This provided at least one date extracted from an external reference for 93% of CVEs, with multiple dates extracted for 73% of CVEs. To confirm the soundness of this strategy, we randomly sampled 100 crawled pages, finding all relevant dates were correctly extracted.

We estimate the earliest disclosure date as the earliest amongst the extracted reference dates and the CVE publication date. While this is a best-effort approach, we observe that it yields *significantly* improved disclosure estimation. Figure 2 plots the CDF of the number of days the estimated disclosure date precedes the CVE publication date. For approximately 8% of CVEs, we did not extract an earlier external reference date, resulting in no improvement for disclosure estimation. However, the median difference is nearly a month (27 days). At the extreme, we witness differences on the order of years. These correspond to vulnerabilities that are assigned CVE IDs and publicly disclosed, but are not published to the NVD until much later. For example, CVE-2013-4119 is a vulnerability in FreeRDP that was first discussed on an OpenWall mailing list in July, 2013 and assigned a CVE ID. However, its NVD entry was not published until October, 2016, resulting in a large discrepancy between the CVE publication date and the true disclosure date. Thus, our method provides us with significantly improved disclosure date estimates.

3.6 Limitations

Vulnerability databases (VDBs) can provide rich sources of data for analysis of security issues and fixes. However, we must bear in mind a number of considerations when using them:

Vulnerability Granularity: By relying on the NVD, we can only assess vulnerabilities at CVE ID granularity. While CVE IDs are widely used, alternative metrics exist for determining what qualifies as a distinct vulnerability [10].

Completeness: No VDB is complete, as they all draw from a limited set of sources. However, by using a VDB as expansive as the NVD, we aim for our analysis to provide meaningful and generalizable insights into vulnerabilities and security fixes.

Quality: The NVD data is manually curated and verified when a vulnerability is assigned a CVE ID, which ideally improves the data quality. However, given the sheer number of vulnerabilities reported, the NVD may contain errors. Throughout our analysis, we aim to identify and investigate anomalous data as part of our methodology for reducing the impact of faulty information.

Source Bias: A VDB may be biased towards certain vulnerabilities or types of software, depending on their vulnerability data sources. Given the extensive range of software considered by the NVD, we anticipate that our findings will remain largely applicable to open-source software.

Reporting Bias: Security researchers may exhibit bias in what security issues they investigate and report, potentially affecting a VDB's set of vulnerabilities. For example, researchers may focus more on publishing high-severity issues, rather than low impact, hard-to-exploit vulnerabilities. Additionally, researchers may favor investigating certain vulnerability types, such as SQL injections or buffer overflows. As a result, we can find raw vulnerability counts ineffective for comparing trends in the security status of software, and we avoid drawing conclusions from such analysis.

In addition to the above considerations, our data collection methodology introduces bias towards open-source software projects, particularly those using Git for versioning. Thus, our findings might not directly apply to other software systems, such as closed-source ones. However, our dataset does provide a diverse sample of 682 software projects.

Finally, our methodology and analyses do rely on some approximations. With a diverse dataset of different types of vulnerabilities across numerous projects, we argue that approximations will often prove necessary, as more accurate metrics would require perhaps intractable levels of manual effort. For example, evaluating a vulnerability's life span requires understanding the context about the vulnerability type and the code logic. An automated approach, if feasible, likely still requires developing a different method for each vulnerability class, and perhaps each type of project. Prior case studies [19, 20, 28] that considered vulnerability life spans relied on manual identification of vulnerability introduction, limiting their scope of investigation. When we do use approximations, we use conservative methods that provide upper/lower bounds in order to still obtain meaningful insights. However, we acknowledge that these bounds may not fully reflect observed effects or properties.

4 DATA CHARACTERIZATION

In this section, we explore the characteristics of the selected CVEs and the collected Git software repositories.

4.1 Vulnerability Publication Timeline

In total, we collected 4,080 security fixes for 3,094 CVEs (implying multiple security fixes for some CVEs, an aspect we explore further in Section 5.1.3). The earliest CVE with a collected security patch was published on August 4, 2005, and the most recent on December 20, 2016. In Figure 3, we plot the timeline of these CVEs, bucketed by the publication month. We observe that our CVE dataset spans this 11 year period, although it exhibits skew towards more recent vulnerabilities. Note that, as discussed in Section 3.6, these raw counts do not imply that our studied software projects have become more vulnerable over time. Rather the increase may reflect other factors such as additional reporting by security researchers.

4.2 Affected Software Products

The NVD also enumerates software products affected by a particular vulnerability for all CVEs in our dataset. We observe a long tail of 856 distinct products, with the top 10 listed in Table 1. The number of products affected exceeds the number of software projects we collected because a CVE vulnerability in one project can affect multiple products that depend on it. Similarly we note that many of

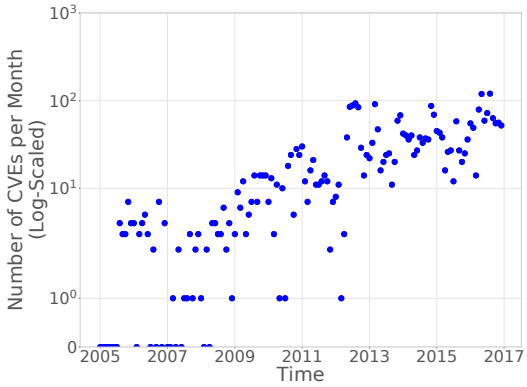


Figure 3: Timeline of CVEs with collected security fixes, grouped by publication month.

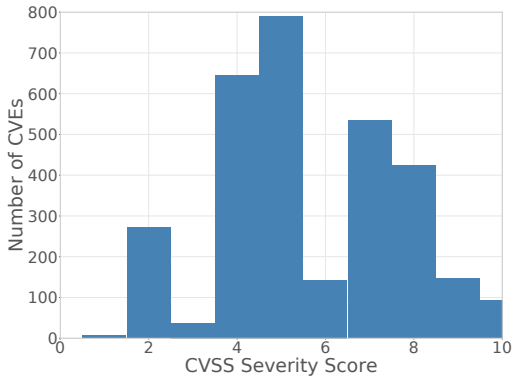


Figure 4: Distribution of CVSS severity scores, which are on a scale of 0 to 10, rounded to the nearest integer.

	Top Products	Num. CVEs
1.	Linux Kernel	917
2.	Ubuntu	211
3.	FFmpeg	187
4.	Debian	170
5.	Wireshark	146
6.	openSUSE	134
7.	PHP	125
8.	Android	121
9.	Fedora	105
10.	QEMU	77

Table 1: The top 10 software products by the number of associated CVE IDs.

	CWE ID	Weakness Summary	Num. CVEs
1.	119	Buffer Overflow	539
2.	20	Improper Input Validation	390
3.	264	Access Control Error	318
4.	79	Cross-Site Scripting	273
5.	200	Information Disclosure	228
6.	189	Numeric Error	221
7.	399	Resource Management Error	219
8.	362	Race Condition	72
9.	89	SQL Injection	61
10.	310	Cryptographic Issues	42

Table 2: Top 10 CWE software weaknesses by the number of CVEs.

the top affected products are Linux distributions, as a vulnerability that affects one distribution frequently occurs in others. This bias in our CVE dataset towards Linux-related vulnerabilities informs us of the importance of per-repository analysis, in addition to aggregate analysis over all CVEs. Such analysis equally weighs the influence of each software project on any computed metrics.

4.3 Vulnerability Severity

The NVD quantifies the severity of vulnerabilities using a standardized method called CVSS (version 2) [12, 35]. While the CVSS standard is imperfect [25], it provides one of the few principled ways to characterize vulnerability risk and potential impact. We use this score as is, however acknowledging the difficulties in objectively assessing vulnerability severity.

All CVEs in our dataset are assigned CVSS severity scores, ranging from 0 to 10. In Figure 4, we depict the distribution of CVSS severity scores for these vulnerabilities, rounded to the nearest integer. These scores reflect the severity of the vulnerability, with 0 to 3.9 deemed low severity, 4.0 to 7.9 labeled medium, and 8.0 to 10.0 regarded as highly severe. We observe that the NVD data consists of vulnerabilities ranging across all severity scores. However, there is a substantial skew towards medium and high scores, which may be the visible effect of security researchers favoring reports of higher-value vulnerabilities (related to the limitations outlined in Section 3.6).

4.4 Vulnerability Categories

The Common Weakness Enumeration (CWE) is a standard for identifying the class of software weaknesses that resulted in a particular security issue [24]. The final NVD annotation we consider is the vulnerability's CWE identifiers, indicating the vulnerability categories. A CWE ID is assigned for 87% of CVEs in our dataset. In total, there are 45 unique CWE IDs associated with our vulnerabilities. Table 2 enumerates the most common software weaknesses, including frequent security problems such as buffer overflows and cross-site scripting errors. However, again we observe that our vulnerabilities span a wide variety of security issues.

4.5 Vulnerability Distribution over Repositories

Our selected CVE vulnerabilities were unevenly distributed over 682 Git projects, as visible in Figure 5. Our dataset contains one vulnerability for the majority of projects, and a heavy skew towards a smaller set of projects (e.g., the Linux kernel has over 900 CVE-related commits). Due to this skew, our analysis must consider per-repository averages, in addition to aggregates.

Figure 5 also illustrates the total number of commits in repository logs. We see that our repositories have varying levels of development, ranging from 3 commits for the "Authoring HTML" Drupal module to over 100,000 commits for projects such as the Linux kernel, LibreOffice, MySQL server, and the PHP interpreter.

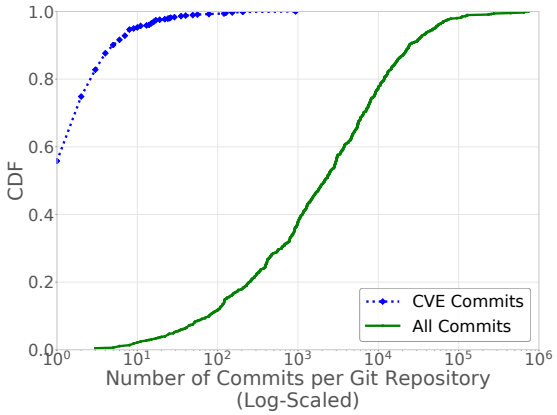


Figure 5: CDFs of the number of CVE commits and all commits for our collected Git repositories.

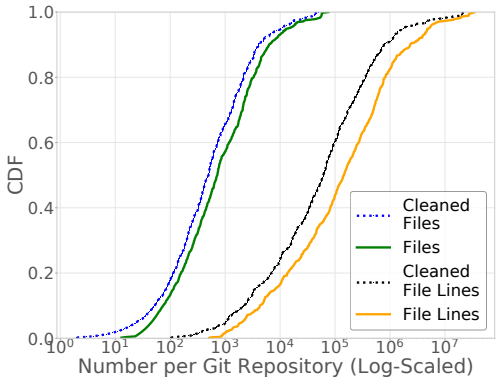


Figure 6: CDFs of the number of files and file lines in our collected repositories and their cleaned versions.

4.6 Repository Size

We can characterize a repository's size by the number of files it has, or the number of lines in those files. In Figure 6, we plot the CDFs of these metrics, for both the original repositories and their cleaned versions (as described in Section 3.4). Our selected projects vary widely in their sizes along both metrics. We find small projects affected by vulnerabilities, such as the SQL injection bug (CVE-2013-3524) in phpVMS's "PopUpNews" module, consisting of 4 PHP files with 103 lines of code. On the other extreme, the Linux kernel contains 20 million lines of code across 44,000 files.

5 ANALYSIS RESULTS

Our collected dataset consists of a diverse set of security vulnerabilities across numerous software projects, for which we have downloaded the source code repositories and amassed a set of both security and non-security bug fixes. The combination of the metadata about patched vulnerabilities and the direct visibility into the corresponding source codes (as well as their history of changes) affords us with a unique perspective on the development life cycle of security fixes, as well as on the characteristics of the security patches themselves (in comparison to non-security bug fixes). In this section, we discuss our corresponding analysis and findings.

When exploring differences between two groups, we determine the statistical significance of our observations using permutation tests with 1,000 rounds. For each group we use a summary statistic of the area under the CDF curve for the investigated metric. In each round of a permutation test, we randomly reassign group labels to all data points (such that group sizes remain constant), recompute the summary statistic for each group, and determine if the summary statistic difference between the newly formed groups exceeds that of the original groups. If the null hypothesis holds true and no significant difference exists between the groups, then the random permutation will only reflect stochastic fluctuations in the summary statistic difference. We assess the empirical probability distribution of this measure after the permutation rounds, allowing us to determine the probability (and significance) of our observed differences. We compute all of the reported p-values via this approach, and using a significance threshold of $\alpha = 0.05$.

5.1 Patch Development Life Cycle

From a software project's perspective, a vulnerability advances through several events throughout its life, such as its introduction into the code base, its discovery and the subsequent patch development, the public disclosure of the security issue, and the distribution of the fix. Prior studies have analyzed the vulnerability life cycle from a public perspective [14, 27, 32], observing when a vulnerability became disclosed to the public and when the corresponding patch was publicly distributed. However, these works have not delved into the project developer side of the remediation process and the life cycle of the patch development itself. Such an exploration can help illuminate the responsiveness of developers to patching vulnerabilities, how long fixes are available before they are actually distributed publicly, and how successfully developers resolve security issues. Here, we investigate the patch development process by connecting the vulnerability information available in the NVD with the historical logs available in Git repositories.

5.1.1 Vulnerability Life Spans in Code Bases. Upon a vulnerability's first discovery, we might naturally ask how long it plagued a code base before a developer rectified the issue. We call this duration the vulnerability's *code base life span*—a notion distinct from the vulnerability's *window of exposure* as investigated in prior work [14, 32], which measures the time from the first release of a vulnerable software version to the public distribution of its patch. As the development and distribution of a patch often occur at different times (a factor we explore in Section 5.1.2), the code base life span reflects the window of opportunity for attackers who silently discover a vulnerability to leverage it offensively, before any defensive measures are taken.

Reliably determining when a vulnerability was born in an automated fashion is difficult, as it requires semantic understanding of the source code and the nature of the vulnerability. However, we can approximate a *lower bound* on age by determining when the source code affected by a security fix was previously last modified. We note that this heuristic does assume that security fixes modify the same lines that contained insecure code, which may not always be the case. However, we assessed whether this is a

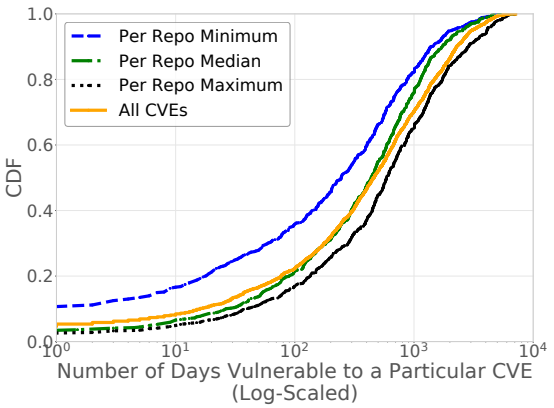


Figure 7: CDFs of CVE vulnerability life spans, for all CVEs and when grouped by software repositories.

robust approximation by randomly sampling 25 security patches. We observed that only 1 did not touch an originally insecure code region, enabling us to conclude that the vast majority of security fixes do modify the culprit code regions.

We analyzed the cleaned versions of security commit data to focus on source code changes. For all lines of code deleted or modified by a security commit, we used Git's "blame" functionality to retrieve the last time each line was previously updated (the blame date).[4] We conservatively designate the most recent blame date across all lines as the estimated date of vulnerability introduction. Then, the duration between this date and the commit date provides a lower bound on the vulnerability's code base life span.

How long do vulnerabilities live in code bases? Figure 7 illustrates the distribution of the lower bound estimates for vulnerability life spans. We plot the distribution for the aggregate of all CVEs, conservatively using the shortest life span for CVEs with multiple commits. To consider potential bias introduced by the uneven distribution of CVEs across repositories (discussed in Section 4.5), we also group commits by their repositories and plot the distributions of the minimum, median, and maximum life span per repository. The aggregate CVE distribution largely follows that of the per-repository median, although it exhibits skew towards longer life spans.

We observe that vulnerabilities exist in code bases for extensive durations. Looking at per-repository medians, we see that 50% had life spans exceeding 438 days (14.4 months). Furthermore, a quarter of repository medians and a third of all CVEs had life spans beyond three years. The longest surviving vulnerability was CVE-2015-8629 in the Kerberos 5 project, patched in January, 2016. The information disclosure vulnerability was first introduced over *21 years ago*.

We observe that 6.5% of our CVEs had a life span lower bound of less than 10 days. Manual inspection identified these as cases where our lower bound was overly conservative, as the vulnerability was introduced at an earlier date. Recent commits happened to touch the same area of code involved in the security fix, resulting in our under-approximation.

[4] Note that we cannot similarly process newly added lines, as they did not exist prior to the commit. We ignore the 22.8% of commits with only additions.

	Weakness Summary	Median Life Span
1.	SQL Injection	230.0
2.	Cross-Site Scripting	290.0
3.	Improper Input Validation	350.0
4.	Access Control Error	373.0
5.	Cryptographic Issues	456.0
6.	Resource Management Error	480.0
7.	Information Disclosure	516.5
8.	Race Condition	573.0
9.	Numeric Error	659.5
10.	Buffer Overflow	781.0

Table 3: Median vulnerability life span in days for the top 10 software weakness categories, as classified by CWE.

Our results concur with prior findings that vulnerabilities live for years, generalized across numerous types of software. Manual evaluation of Ubuntu kernel vulnerabilities [19, 20] found that the average vulnerability's code base life span was approximately 5 years. Similarly, Ozment and Schechter [28] manually analyzed vulnerabilities in OpenBSD, finding the median vulnerability lifetime exceeded 2.6 years, although they noted that OpenBSD emphasizes secure coding practices. We observe that our typical life span estimates are lower than these previous ones, which may be due to our consideration of software projects beyond Linux variants, or our conservative approximation method.

Do more severe vulnerabilities have shorter lives? One might hypothesize that more severe vulnerabilities reside in code bases for shorter periods, as their more visible impact may correlate with more likely discovery and quicker remediation. To explore this aspect, we correlate CVSS severity scores with life spans, computing a Spearman's correlation coefficient of $\rho = -0.062$. This indicates that there is *no* substantial (monotonic) correlation between a vulnerability's severity and its life span. Even if developers are more motivated to remedy severe vulnerabilities, their expediency pales in comparison to the time scale of the initial vulnerability discovery, which our analysis shows is uncorrelated with severity. We note this generalizes an observation that Ubuntu vulnerability life spans likewise did not correlate with severity [20].

Do different types of vulnerabilities have varying life spans? Different classes of vulnerabilities may exhibit varying life spans, as some vulnerabilities might prove more challenging to uncover. In Table 3, we summarize the vulnerability life spans for CVEs exhibiting the top 10 software weaknesses as classified by CWE (as discussed in Section 4.4). We observe that vulnerability life spans vary widely based on the software weakness class. Web-oriented vulnerabilities like SQL injection and cross-site scripting have significantly shorter life spans compared to errors in software logic and memory management. In comparison, race conditions, numeric errors, and buffer overflows remain undiscovered for two to three times as long. (Balancing across software repositories did not change the findings.) We conjecture that the life span variation across different vulnerability types results from both the type of software affected and the nature of the vulnerability. For example, web-oriented issues may appear on websites visited by thousands of users, increasing the likelihood that some problematic scenario arises that uncovers the vulnerability. Also, certain vulnerabilities

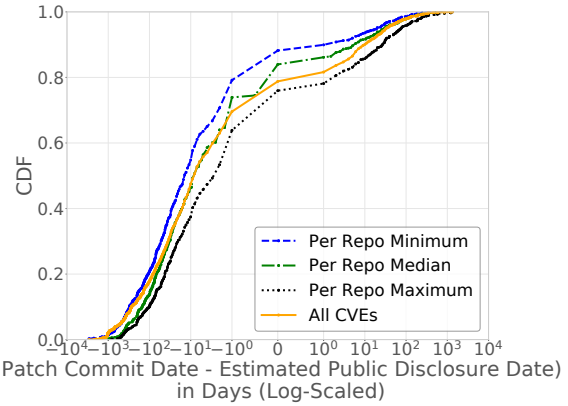


Figure 8: CDFs of the number of days between a vulnerability's public disclosure and its fix, plotted for all CVEs and grouped by software repositories.

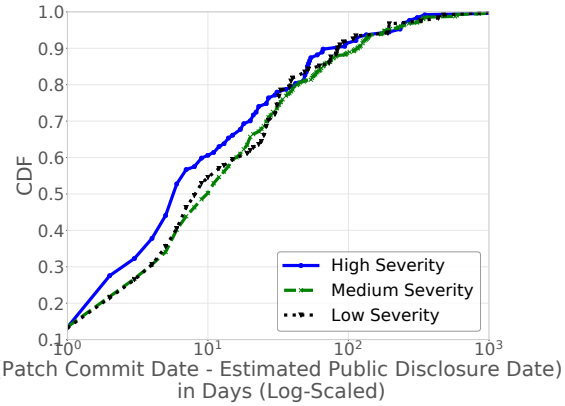


Figure 9: CDFs of the number of days after public disclosure until a CVE has a patch committed, grouped by the CVE severity class.

such as cross-site scripting and SQL injection may be isolated to a small portion of code where reasoning about and identifying issues is more straightforward (compared to other problems such as race conditions).

5.1.2 Security Fix Timeliness. The timeliness of a security fix relative to the vulnerability's public disclosure affects the remediation process and the potential impact of the security issue. On the one hand, developers who learn of insecurities in their code base through unanticipated public announcements have to quickly react before the attackers leverage the information for exploitation. On the other hand, developers who learn of a security bug through private channels can address the issue before public disclosure, but the available patch may not be released for some time due to a project's release cycle, expanding the vulnerability's window of exposure.

We explore this facet of remediation by comparing the patch commit date for CVEs in our dataset with public disclosure dates (estimated as described in Section 3.5). We note that disclosures are not necessarily intertwined with patch releases, although this is the case for the majority of disclosures [14]. In Figure 8, we depict the CDFs of the number of days between disclosure and patching. We plot this for all CVEs, using the earliest patch commit date if a CVE has multiple commits associated with it. We additionally group CVEs by their software repositories, and plot the distribution across repositories. Here, we observe that the aggregate distribution over all CVEs largely matches the distribution over per-repository medians, although the per-repository medians exhibit a slight skew towards smaller absolute values.

How frequently are vulnerabilities unpatched when disclosed? In Figure 8, vulnerabilities publicly disclosed but not yet fixed manifest as positive time difference values. This occurred for 21.2% of all CVEs. We cannot determine whether these vulnerabilities were privately reported to project developers but with no prior action taken, or disclosed without any prior notice. However, over a quarter (26.4%) of these unpatched security issues remained unaddressed 30 days after disclosure, leaving a window wide open for attacker exploitation. This generalizes the observation made by Frei [13], who found that approximately 30% of Windows vulnerabilities were unpatched at disclosure and some remained so for over 180 days.

How frequently are vulnerabilities fixed by disclosure time? The predominant behavior in Figure 8, occurring for 78.8% of all CVEs, is that the security fixes were committed by public disclosure time, manifesting as negative or zero time differences. This suggests that the majority of vulnerabilities were either internally discovered or disclosed to project developers using private channels, the expected best practice.

Are vulnerability patches publicly visible long before disclosure? From Figure 8, we see that nearly 70% of patches were committed before disclosure (having negative time difference values). The degree to which security commits precede disclosures varies widely, which upon manual inspection appears to arise due to the different release cycles followed by various projects (and variations within each project's development timeline). This behavior highlights the security impact of an interesting aspect of the open-source ecosystem. Open-source projects are not frequently in a position to actively distribute security updates. Rather, we observe that projects roll security fixes into periodic version releases that users must directly download and install, or updates are pulled downstream for incorporation by software distribution platforms (such as package repositories maintained by Linux OS variants). Announcements about the releases or updates, and the security fixes they contain, follow shortly after.

Unfortunately, this development and deployment process also provides a window of opportunity for exploitation. Given the public nature of open-source projects and their development, an attacker targeting a specific software project can feasibly track security patches and the vulnerabilities they address. While the vulnerability is addressed in the project repository, it is unlikely to be widely fixed in the wild before public disclosures and upgrade distribution. From Figure 8, we note that over 50% of CVEs were patched more than a week before public disclosure, giving attackers ample time to develop and deploy exploits.

Are higher severity vulnerabilities patched quicker? All vulnerabilities are not equal, as they vary in exploitation complexity and requirements, as well as security impact. One might expect these factors to affect the patch development process, as developers may prioritize fixing certain vulnerabilities over others. To explore whether a vulnerability's severity (scored using CVSS) affects patch timeliness behavior, we cluster CVEs by their severity categories (low, medium, and high). We find that severity significantly affects whether a fix is developed before or after public disclosure. 88.1% of high severity CVEs were patched prior to public announcements, compared to 78.2% of medium severity bugs and 58.8% of low severity issues. These differences indicate that project developers prioritize higher impact vulnerabilities when determining if and when to address them.

While one might also expect earlier disclosures for more severe vulnerabilities, we observe no significant differences ($p > 0.12$) across severity categories when investigating the time by which a patch precedes disclosure (for vulnerabilities fixed by disclosure time). This fits with the common model used by many open-source projects of rolling security patches (of all severity levels) into recurrent releases and announcements. When exploring the time after disclosure until patching (for vulnerabilities unpatched at disclosure), we find that highly severe vulnerabilities get patched more quickly, as shown in Figure 9. This difference is significant ($p < 0.013$), indicating project developers respond quicker to more serious disclosed-yet-unpatched vulnerabilities.

5.1.3 Patch Reliability. The patch a developer creates to address a vulnerability may unfortunately disrupt existing code functionality or introduce new errors. Beyond the direct problems that arise from such patches, end user trust in generally applying patches (or in the software itself) can erode. To assess how successful developers are at producing reliable and safe security fixes, we attempted to identify instances of multiple commits for the same CVE, and classify the causes.

How frequently are security patches broken (e.g., incomplete or regressive)? In total, 11.5% of CVEs had multiple associated commits for a single repository in the NVD data. However, if an initial patch introduced an error or was incomplete, the NVD entry might not have been updated with the follow-on fix. After the NVD entry is published, NVD analysts are unlikely to continue tracking a CVE unless new updates are reported to them. Thus, we attempted to identify further commits that may be associated with a CVE using repository Git logs.

For each security patch commit and its commit hash H, we searched the repository's Git log for any other commits that had a commit message including the CVE ID or the 7-character prefix of the commit hash. We considered this prefix as it is used as the Git short version of the commit hash, and matches any longer hash prefixes. This method finds related commits which were not distinct patches, such as merge, rebase, and cherry-pick commits. To filter these, we ignored commits with diffs identical to an earlier one, and commits with multiple parents (e.g., merges). Note that we could only identify multiple patches when commit messages contained this explicit linkage, so our analysis provides a lower bound.

Using this approach, we identified a total of 682 CVEs with multiple commits, 22.0% of all CVEs. Not all multi-commit fixes are necessarily problematic though, as project developers may split a fix into multiple commits that they push to the repository in close succession. We observed that 242 CVEs had all fixes committed within 24 hours. Given the limited time window for potential newly introduced problems, we designate these as *piecewise* fixes and non-problematic.

CVE Commits Label	Num. CVEs	Median Num. Follow-On Commits	Median Fix Inter-Arrival Time
Incomplete	26 (52%)	1.0	181.5 Days
Regressive	17 (34%)	1.0	33.0 Days
Benign	14 (28%)	1.5	118.5 Days

Table 4: Summary of our manual investigation into 50 randomly sampled CVEs with multiple commits. Note that a CVE may have commits in multiple categories. Follow-on commits include all commits associated with the original patch.

We randomly sampled 50 of the remaining 440 CVEs and manually investigated if the fixes were problematic. Table 4 summarizes our results. We identified 26 (52%) as *incomplete*, where the initial fix did not fully patch the vulnerability, requiring a later patch to complete the job. We labeled 17 (34%) as *regressive*, as they introduced new errors that required a later commit to address. The overlap included 4 CVEs (8%) with both incomplete and regressive patches. Other follow-on fixes were *benign*, such as commits for added documentation, testing, or code cleanup/refactoring. 11 CVEs (22%) had only these benign additional commits (although 3 other CVEs had both benign and problematic commits). Note that our random sample was not biased towards any particular project, as it spanned 42 repositories.

Extrapolating from the random sample to the remaining 440 CVEs with non-piecewise multiple commits (accounting for 14.2% of all CVEs), we estimate that about 7% of *all* security fixes may be incomplete, and about 5% regressive. These findings indicate that broken patches occur unfortunately frequently, and applying security patches comes with non-negligible risks. In addition, these numbers have a skew towards underestimation: we may not have identified all existing problematic patches, and recent patches in our dataset might not have had enough time yet to manifest as ultimately requiring multiple commits.

We note that our observed frequency of failed security fixes is similar to or lower than that observed by prior studies on general bug fixes. Gu et al. [17] observed that 9% of bug fixes were bad across three Java projects while Yin et al. [40] found that between 15%–25% of general bug patches for several Linux variants were problematic. As our detection of problematic security fixes skews towards underestimation, it is undetermined whether security fixes are more or less risky than other bug fixes. However, it is clear that security patches do suffer failures similarly to non-security bug fixes.

How long do problematic patches remain unresolved? As shown in Table 4, for both incomplete and regressive patches in our sample, we find the median number of additional patches required to rectify the original broken patches to be only one commit. The

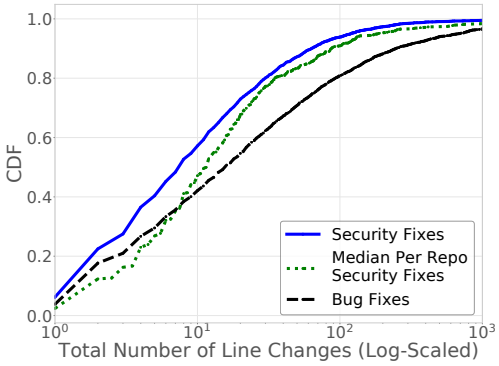


Figure 10: CDFs of the total number of line changes, for all security and non-security bug fixes, and the median of security commits grouped by repository.

typical incomplete fix takes *half a year* to remedy, and patches problematic enough to require reverting typically take a month to repair. Thus, problematic security patches can remain unresolved for extensive durations of time.

5.2 Patch Characteristics

While numerous works have investigated general software patches [29, 33, 34, 42], few have considered what distinguishes security patches from other non-security bug fixes. Intuitively, the software conditions resulting, for example, in buffer overflow and SQL injection vulnerabilities can differ greatly from those that produce performance and concurrency bugs. Thus, the characteristics of their fixes may likewise prove different. Indeed, Zama et al. [41] conducted a case study on security and performance fixes for Mozilla Firefox, observing differences in the remediation for the two bug types. These characteristics are important to understand as they may reflect our ability to expeditiously generate patches, verify their safety, or assess their impact on applications. Here, we compare our collection of security and non-security bug fixes to help illuminate their differences, considering facets such as the complexity of fixes and the locality of changes.

5.2.1 Non-Source Code Changes. **Do security and non-security bug fixes always modify source code?** Given the nature of bug fixes, one might expect them to universally involve source code changes. We explore this hypothesis by contrasting our commit data with their cleaned versions (source code comments and non-source code files removed). We find that the hypothesis does not hold: a non-trivial fraction of commits involved no code changes. For non-security bug fixes, 6.1% involved erroneous configurations, build scripts with broken dependencies or settings, incorrect documentation, and other non-source code changes.

More surprising, we find that 1.3% of security fixes also did not touch source code. In some cases, the commit added a patch file to the repository without applying the patch to the code base. However, numerous CVE vulnerabilities *do not* reside in the source code. For example, CVE-2016-7420 was assigned to the Crypto++ Library for not documenting the security-related compile-time requirements, such that default production builds may suffer information disclosure vulnerabilities. Similarly, the fix for CVE-2016-3693 involved changing a project library dependency to a new version, as the inclusion of the older versions allowed attackers to access sensitive system information.

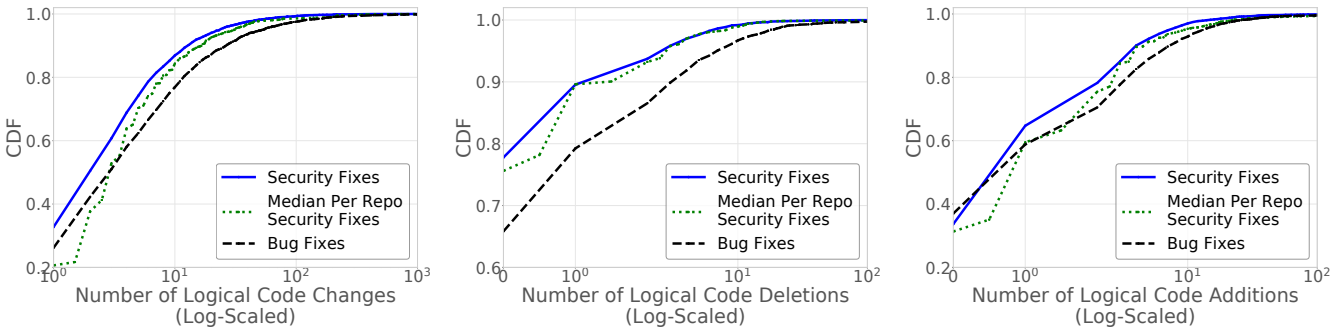
Thus, bug fixes are not exclusively associated with source code modifications, although this is significantly more likely with non-security bug fixes than with security patches. For further analysis on commit characteristics, we focus on the cleaned versions, excluding the commits that did not modify code.

5.2.2 Patch Complexity. How complex are security patches compared to other non-security bug fixes? We can assess software complexity using various metrics, although some, such as cyclomatic complexity [22], require deep analysis of a code base. Given the number and diversity of software projects we consider, we chose lines of code (LOC) as a simple-albeit-rudimentary metric, as done in prior studies [18, 26, 33, 41].

Are security patches smaller than non-security bug fixes? In Figure 10, we plot the CDFs of the total LOC changed in cleaned commit diffs, for all security and non-security patches, as well as the median security fix per repository. This conservative metric sums the LOC deleted or modified in the pre-commit code with those modified or added post commit, providing an upper bound on the degree of change. We see that compared to per-repository medians, the aggregate of security commits skews towards fewer total LOC changed. Under this metric, security commits overall are statistically significantly less complex and smaller than non-security bug patches ($p \approx 0$). The median security commit diff involved 7 LOC compared to 16 LOC for non-security bug fixes. Approximately 20% of non-security patches had diffs with over 100 lines changed, while this occurred in only 6% of security commits. When considering per-repository medians, our conclusions differ only slightly, in that non-security bug fixes have a slightly larger portion of very small commits with diffs less than 9 LOC, but are typically larger.

Do security patches make fewer "logical" changes than non-security bug fixes? As an alternative to our raw LOC metric, we can group consecutive lines changed by a commit as a single "logical" change. Under this definition, several lines updated are considered a single logical update, and a chunk of deleted code counts as a single logical delete. We depict the CDFs of the number of logical actions per commit in Figure 11, although we omit a plot for logical updates as it closely resembles that of all logical changes. In all cases, we observe that per-repository medians skew less towards very small numbers of logical actions compared to security commits in aggregate. Across all logical actions, we observe that security commits involve significantly fewer changes (all $p < 0.01$). Nearly 78% of security commits did not delete any code, compared to 66% of non-security bug-fix commits. Between 30% to 40% of all commits also did not add any code, indicating the majority of logical changes were updates.

Do security patches change code base sizes less than non-security bug fixes? Another metric for a patch's complexity is its impact on the code base size. The net number of lines changed by a commit reflects the growth or decline in the associated code base's size. In Figure 12, we plot the CDFs of these size changes. We observe that significantly more non-security bug patches result



(a) All Logical Changes **(b) Logical Deletes** **(c) Logical Additions**

Figure 11: CDFs of the number of logical changes introduced by a commit, for all security and non-security bug fixes, and for the median amongst security commits grouped by repository. We omit a plot for logical updates, which looks very similar to that for all logical changes because logical updates predominate. Note the varying minimum y-axis values.

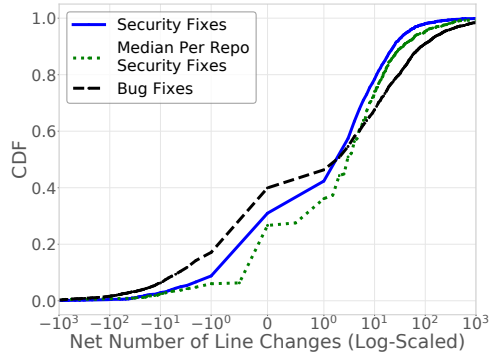


Figure 12: CDFs of the net number of line changes, for all security and non-security patches, and the median of security commits grouped by repository.

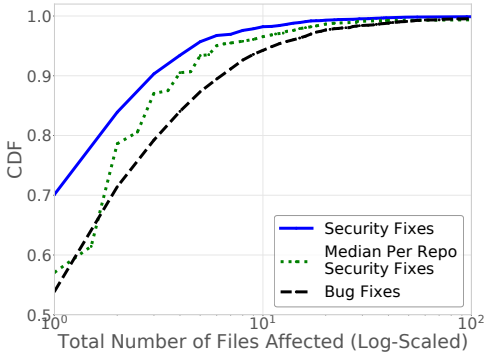


Figure 13: CDFs of the number of files affected, for all security and non-security bug fixes, and the median of security fixes grouped by repository.

in a net reduction in project LOC, compared to security fixes: 18% of non-security bug fixes reduced code base sizes compared to 9% of security patches. For all commits, approximately a quarter resulted in no net change in project LOC, which commonly occurs when lines are only updated. Overall, projects are more likely to grow in size with commits, as the majority of all commits added to the code base. However, security commits tend to contribute less growth compared to non-security bug fixes, an observation that accords with our earlier results.

These findings support the notion that security fixes are generally less complex than other bug fixes. We note that this generalizes the same conclusion drawn for Mozilla Firefox by Zama et al. [41].

5.2.3 Commit Locality. Finally, we can quantify the impact of a patch by its *locality*. We consider two metrics: the number of files affected and the number of functions affected.

Do security patches affect fewer source code files than non-security bug fixes? Figure 13 illustrates the CDFs of the number of files touched by fixes. From this, we see that security patches modify fewer files compared to non-security bug fixes, a statistically significant observation ($p \approx 0$). In aggregate, 70% of security patches affected one file, while 55% of non-security bug patches were equivalently localized. Fixes typically updated, rather than created or deleted, files. Only 4% of security fixes created new files (vs. 13% of non-security bug fixes), and only 0.5% of security patches deleted a file (vs. 4% of non-security bug fixes).

Do security patches affect fewer functions than non-security bug fixes? To pinpoint the functions altered by patches, we used the `ctags` utility [4] to identify the start of functions in our source code. We determined the end of each function under the scoping rules of the corresponding programming language, and mapped line changes in our commit diffs to the functions they transformed. Figure 14 shows the CDFs of the number of functions affected by patches. We find that 5% of non-security bug fixes affected only global code outside of function boundaries, compared to 1% of security patches. Overall, we observe a similar trend as with the number of affected files. Security patches are significantly ($p \approx 0$) more localized across functions: 59% of security changes resided in a single function, compared to 42% of other bug fixes.

In summary, our metrics indicate that security fixes are more localized in their changes than other bug fixes.

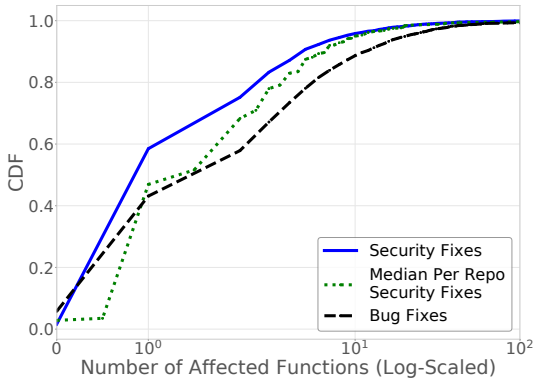


Figure 14: CDFs of the number of functions modified, for all security and non-security bug patches, and the median of security fixes grouped by repository.

6 DISCUSSION

In this study, we have conducted a large-scale empirical analysis of security patches across over 650 projects. Here we discuss the main takeaways, highlighting the primary results developed and their implications for the security community moving forward.

Need for more extensive or effective code testing and auditing processes for open-source projects: Our results show that vulnerabilities live for years and their patches are sometimes problematic. Using a lower bound estimation method, our exploration of vulnerability life spans revealed that over a third of all security issues were first introduced more than 3 years prior to remediation. The issues do not cease once a vulnerability is first addressed; almost 5% of security patches negatively impacted the software, and over 7% were incomplete and left the security hole present.

These findings indicate that the software development and testing process, at least for open-source projects, is not adequate at quickly detecting and properly addressing security issues. There are several important implications due to these shortcomings. An attacker who discovers a zero-day vulnerability can retain its viability with reasonable confidence for on the order of years. While large-scale exploitation of a zero-day may result in its detection and subsequent remediation, targeted attacks may persist unnoticed. Similarly, a subtle backdoor inserted into a code base will also likely survive for a prolonged period, with only commit code reviews (if performed) as the final barrier. The not infrequent occurrences of broken security patches also have negative implications on user patching behavior. Applying a patch has often been viewed as risky, and negative experiences with problematic updates (particularly regressive ones) can drive users away from remedying security bugs in a timely fashion.

A natural avenue for future work is to develop more effective testing processes, particularly considering usability, as developers are unlikely to leverage methods that prove difficult to deploy or challenging to interpret. One example of such research is VCCFinder [30], a code analysis tool that assists with finding vulnerability-introducing commits in open-source projects. In addition, software developers can already make strides in improving their testing processes by using existing tools more extensively. For example, sanitizers such as ASan [15], TSan [15] and UBSan [9] help detect various errors that may result in security bugs. Fuzzers (such as AFL [1]) also assist in identifying inputs that trigger potentially exploitable issues.

The transparency of open-source projects makes them ripe for such testing not only by the developers, but by external researchers and auditors as well. Community-driven initiatives, such as those supported by the Core Infrastructure Initiative [3], have already demonstrated that they can significantly improve the security of open-source software. For example, the Open Crypto Audit Project [8] audited the popular encryption software TrueCrypt, while Google's OSS-Fuzz program [16] offers continuous fuzzing of critical open-source infrastructure for free, already discovering and reporting hundreds of bugs. Further support of such efforts, and more engagement between various project contributors and external researchers, can help better secure the open-source ecosystem.

Need for refined bug reporting and public disclosure processes for open-source projects: Our analysis of the timeliness of security fixes revealed that they are poorly timed with vulnerability public disclosures. Over 20% of CVEs were unpatched when they were first announced, perhaps sometimes to the surprise of project developers. While we observed that these were more likely to be low-severity vulnerabilities, many were still medium- and high-severity bugs, unfixed for days to weeks post-disclosure. This gap provides attackers with the knowledge and time to strike.

In the opposite direction, we discovered that when security issues are reported (or discovered) privately and fixed, the remedy is not immediately distributed and divulged, likely due to software release cycles. Over a third of fixed vulnerabilities were not publicly disclosed for more than a month. While operating in silence may help limit to a small degree the dissemination of information about the vulnerability, it also forestalls informing affected parties and spurring them to remediate. Given the transparency of open-source projects, attackers may be able to leverage this behavior by tracking the security commits of target software projects (perhaps by training a classifier or keying in on common security-related terms in commit messages). From the public visibility into these commits, attackers can identify and weaponize the underlying vulnerabilities.

However, the open-source nature of projects need not be a liability when patching vulnerabilities. Transparent bug reporting instructions, containing the proper point of contact, the required diagnostic information, the expected remediation timeline, and potential incentives (such as bug bounties or "hall of fame" status), can expedite the vulnerability reporting process. Fixes for vulnerabilities can also be disclosed in better coordination with public disclosures. For example, the Internet Systems Consortium (ISC), maintainer of the open-source DNS software BIND and DHCP implementations, has established explicit disclosure policies that embargo publicly revealing security patches until near public disclosure time [7]. Instead, ISC customers, OEMs, operating system maintainers, and other vendors who re-package ISC open-source software are privately notified about vulnerabilities and their patches prior to public disclosure. A controlled disclosure process informs some of the most heavily affected parties before public disclosure, providing adequate

time to prepare properly, while reducing the leakage of vulnerability information pre-disclosure. Additionally, outreach efforts to notify end-systems affected by a vulnerability have shown some promise [21]. While we advocate that open-source projects should adopt such a disclosure process, they should be transparent about the process itself and execute it consistently, avoiding hasty and uncoordinated disclosures such as with the Heartbleed bug [11].

Opportunities for leveraging characteristics of security patches: Our comparison of security patches with non-security bug fixes revealed that security fixes have a smaller impact on code bases, along various metrics. They involve fewer lines of code, fewer logical changes, and are more localized in their changes. This has implications along various patch analysis dimensions.

Tying back to broken patches, the lower complexity of security patches can perhaps be leveraged for safety analysis customized for evaluating just security fixes. Also, as these remedies involve fewer changes, automatic patching systems may operate more successfully if targeting security bugs. Zhong and Su [42] observed that general patches are frequently too complex or too delocalized to be amenable to automatic generation. However, security patches may be small and localized enough. From a usability angle, we may additionally be able to better inform end users of the potential impact of a security update, given its smaller and more localized changes. The need for more exploration into the verification and automated generation of security patches is quite salient as our ability to respond to security concerns has remained relatively unchanged, while the attack landscape has grown ever more dangerous.

7 CONCLUSION

In this paper, we conducted a large-scale empirical study of security patches, evaluating over 4,000 security fixes across a diverse set of 682 software projects. The investigation centered around a dataset we collected that merges vulnerability entries from the NVD, information scraped from relevant external references, affected source code repositories, and their associated security fixes. Using these disparate data sources, we analyzed facets of the patch development life cycle. In addition, we extensively characterized the security patches themselves, contrasting them with non-security bug fixes.

Our findings have revealed shortcomings in our ability to quickly identify vulnerabilities and reliably address them. Additionally, we have observed that the timing of public disclosure does not closely align with the date a patch is applied to the code base, providing windows of opportunity for attacker exploitation. Our characterization of security fixes shows they are less complex and more localized than other non-security bug fixes, perhaps making them more amenable to software analysis and automatic repair techniques. By leveraging these insights, we hope the security community can progress in improving the remediation process for security vulnerabilities.

ACKNOWLEDGMENTS

We thank Christopher Thompson and Sascha Fahl for helpful feedback on our study. This work was supported in part by the National Science Foundation awards CNS-1237265 and CNS-1518921, for which we are grateful. The opinions expressed in this paper do not necessarily reflect those of the research sponsors.

REFERENCES

[1] American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/.
[2] cgit. https://git.zx2c4.com/cgit/about/.
[3] Core Infrastructure Initiative. https://www.coreinfrastructure.org.
[4] Exuberant Ctags. http://ctags.sourceforge.net/.
[5] GitLab. https://about.gitlab.com/.
[6] GitWeb. https://git-scm.com/book/en/v2/Git-on-the-Server-GitWeb.
[7] ISC Software Defect and Security Vulnerability Disclosure Policy. https://kb.isc.org/article/AA-00861/164/ISC-Software-Defect-and-Security-Vulnerability-Disclosure-Policy.html.
[8] Open Crypto Audit Project. https://opencryptoaudit.org.
[9] Undefined Behavior Sanitizer. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.
[10] Steve Christey and Brian Martin. Buying Into the Bias: Why Vulnerability Statistics Suck. In *BlackHat*, 2013.
[11] Zakir Durumeric, Frank Li, James Kasten, Nicholas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, David Adrian, Vern Paxson, Michael Bailey, and J. Alex Halderman. The Matter of Heartbleed. In *ACM Internet Measurement Conference (IMC)*, 2014.
[12] Forum of Incident Response and Security Teams. Common Vulnerability Scoring System v3.0: Specification Document. https://www.first.org/cvss/specification-document.
[13] Stefan Frei. End-Point Security Failures: Insights gained from Secunia PSI Scans. In *USENIX Predict Workshop*, 2011.
[14] Stefan Frei, Martin May, Ulrich Fiedler, and Bernhard Plattner. Large-Scale Vulnerability Analysis. In *SIGCOMM Workshops*, 2006.
[15] Google. Sanitizers. https://github.com/google/sanitizers.
[16] Google Open Source Blog. Announcing OSS-Fuzz: Continuous Fuzzing for Open Source Software. https://opensource.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html.
[17] Zhongxian Gu, Earl Barr, David Hamilton, and Zhendong Su. Has the Bug Really Been Fixed? In *International Conference on Software Engineering (ICSE)*, 2010.
[18] Zhen Huang, Mariana D'Angelo, Dhaval Miyani, and David Lie. Talos: Neutralizing Vulnerabilities with Security Workarounds for Rapid Response. In *IEEE Security and Privacy (S&P)*, 2016.
[19] Jonathan Corbet. Kernel Vulnerabilities: Old or New?, October 2010. https://lwn.net/Articles/410606/.
[20] Kees Cook. Security Bug Lifetime, October 2016. https://outflux.net/blog/archives/2016/10/18/security-bug-lifetime.
[21] Frank Li, Zakir Durumeric, Jakub Czyz, Mohammad Karami, Michael Bailey, Damon McCoy, Stefan Savage, and Vern Paxson. You've Got Vulnerability: Exploring Effective Vulnerability Notifications. In *USENIX Security Symposium*, 2016.
[22] T. J. McCabe. A Complexity Measure. In *IEEE Transaction on Software Engineering*, 1976.
[23] MITRE Corporation. Common Vulnerabilities and Exposures. https://cve.mitre.org/.
[24] MITRE Corporation. CWE: Common Weakness Enumeration. https://cwe.mitre.org/.
[25] Nuthan Munaiah and Andrew Meneely. Vulnerability Severity Scoring and Bounties: Why the Disconnect? In *International Workshop on Software Analytics (SWAN)*, 2016.
[26] Emerson Murphy-Hill, Thomas Zimmermann, Christian Bird, and Nachiappan Nagappan. The Design of Bug Fixes. In *International Conference on Software Engineering (ICSE)*, 2013.
[27] Antonio Nappa, Richard Johnson, Leyla Bilge, Juan Caballero, and Tudor Dumitras. The Attack of the Clones: A Study of the Impact of Shared Code on Vulnerability Patching. In *IEEE Security and Privacy (S&P)*, 2015.
[28] Andy Ozment and Stuart E. Schechter. Milk or Wine: Does Software Security Improve with Age? In *USENIX Security Symposium*, 2006.
[29] Jihun Park, Miryung Kim, Baishkhi Ray, and Doo-Hwan Bae. An Empirical Study on Supplementary Bug Fixes. In *Mining Software Repositories (MSR)*, 2012.
[30] 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 *ACM Conference on Computer and Communications Security (CCS)*, 2015.
[31] RhodeCode. Version Control Systems Popularity in 2016. https://rhodecode.com/insights/version-control-systems-2016.
[32] Muhammad Shahzad, M. Zubair Shafiq, and Alex X. Liu. A Large Scale Exploratory Analysis of Software Vulnerability Life Cycles. In *International Conference on Software Engineering (ICSE)*, 2012.
[33] Jacek Sliwerski, Thomas Zimmermann, and Andreas Zeller. When Do Changes Induce Fixes. In *Mining Software Repositories (MSR)*, 2005.
[34] Mauricio Soto, Ferdian Thung, Chu-Pan Wong, Claire Le Goues, and David Lo. A Deeper Look into Bug Fixes: Patterns, Replacements, Deletions, and Additions. In *Mining Software Repositories (MSR)*, 2016.

[35] U.S. National Institute of Standards and Technology. CVSS Information. https://nvd.nist.gov/cvss.cfm.
[36] U.S. National Institute of Standards and Technology. National Checklist Program Glossary. https://web.nvd.nist.gov/view/ncp/repository/glossary.
[37] U.S. National Institute of Standards and Technology. National Vulnerability Database. https://nvd.nist.gov/home.cfm.
[38] U.S. National Institute of Standards and Technology. NVD Data Feed. https://nvd.nist.gov/download.cfm.
[39] Zhengzi Xu, Bihuan Chen, Mahinthan Chandramohan, Yang Liu, and Fu Song. SPAIN: Security Patch Analysis for Binaries Towards Understanding the Pain and Pills. In *International Conference on Software Engineering (ICSE)*, 2017.
[40] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. How do Fixes become Bugs? In *ACM European Conference on Foundations of Software Engineering (ESEC/FSE)*, 2011.
[41] Shahed Zaman, Bram Adams, and Ahmed E. Hassan. Security Versus Performance Bugs: A Case Study on Firefox. In *Mining Software Repositories (MSR)*, 2011.
[42] Hao Zhong and Zhendong Su. An Empirical Study on Real Bug Fixes. In *International Conference on Software Engineering (ICSE)*, 2015.

A OBTAINING VULNERABILITY PUBLIC DISCLOSURE DATES

	Domain	Num. References
1.	openwall.com	2413
2.	ubuntu.com	2055
3.	lists.opensuse.org	1784
4.	securityfocus.com	1505
5.	rhn.redhat.com	1328
6.	bugzilla.redhat.com	1158
7.	debian.org	830
8.	lists.fedoraproject.org	673
9.	oracle.com*	573
10.	mandriva.com*	540
11.	vupen.com*	482
12.	xforce.iss.net*	422
13.	marc.info	305
14.	support.apple.com	259
15.	securitytracker.com	235
16.	lists.apple.com	235
17.	seclists.org	204
18.	bugs.wireshark.org	143
19.	bugs.php.net	127
20.	security.gentoo.org	102

Table 5: List of the 20 most common externally referenced sites for CVEs corresponding to our collected security Git commits. We crawled references to these sites for publication dates to better estimate vulnerability public disclosure dates, although not all web pages were still active. Note that 4 sites (marked with asterisks) were no longer active, did not provide publication dates, or employed anti-crawling measures.