A Large-Scale Empirical Study of Security Patches

FRANK LI AND VERN PAXSON

Frank Li is a PhD student at the University of California, Berkeley. His research mainly focuses on improving the remediation process for security issues such as vulnerabilities and misconfigurations. More broadly, he is interested in large-scale network measurements and empirical studies in a computer security context. frankli@cs.berkeley.edu

Vern Paxson is a Professor of Electrical Engineering and Computer Sciences at the University of California, Berkeley, and leads the Networking and Security Group at the International Computer Science Institute in Berkeley. His research focuses heavily on measurement-based analysis of network activity and Internet attacks. He works extensively on high performance network monitoring, detection algorithms, cybercrime, and countering censorship and abusive surveillance. vern@berkeley.edu

Miscreants seeking to exploit computer systems incessantly discover and weaponize new security vulnerabilities. As a result, system administrators and end users must constantly run on the "patch treadmill," where they apply security patch after security patch to fix newly discovered software vulnerabilities, relying on many of the same processes practiced for decades to update their software against the latest threats. Given the vital role that security patches play in our management of vulnerabilities, it behooves us to better understand the patch development process and characteristics of the resulting fixes.

Prior studies [2, 4, 5, 7, 8] investigated aspects of the vulnerability and patching life cycles but typically at a restricted scale in terms of software diversity, focusing on only a few projects or even just one. While these studies provide insights into the patch development process, there remains a question of how generally their findings apply, and how the nature of security patches may differ from that of other types of bug fixes. Security patches are of particular importance given their critical role in securing software and the time sensitivity of their development.

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 data set that merges vulnerability entries from the National Vulnerability Database [6], 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 general bug patches, exploring the complexity of different types of patches and their impact on code bases.

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 metadata. Our data collection centered around the National Vulnerability Database (NVD) [6], a database provided by the US 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.

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

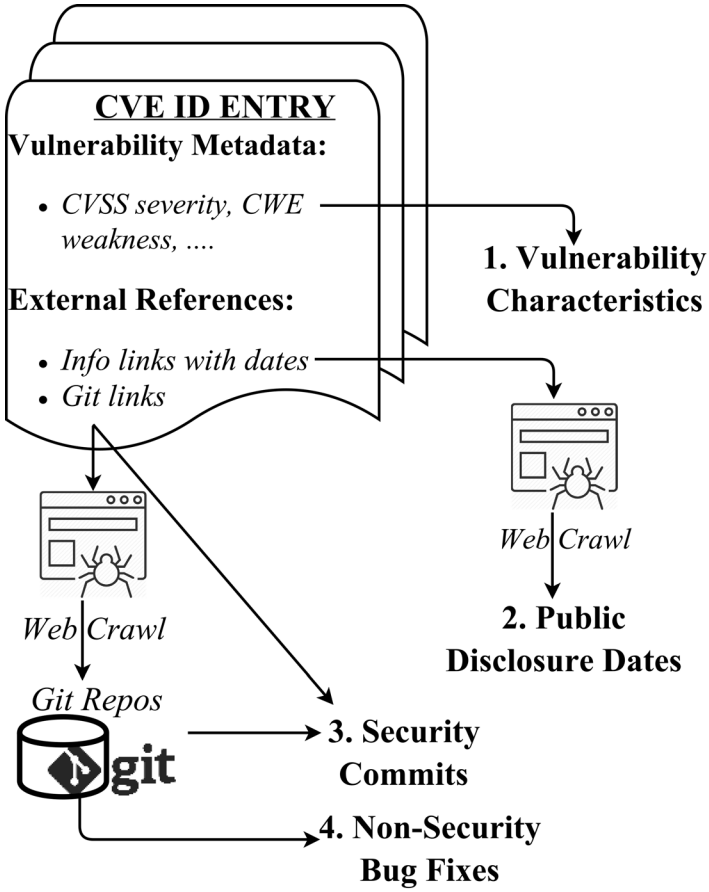

Figure 1: An overview of our data collection methodology. (1) We extracted vulnerability characteristics from CVE entries in the NVD with external references to Git commit links. (2) We crawled other references and extracted page publication dates to estimate public disclosure dates. (3) We crawled the 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 general bug fixes.

of this section, we briefly describe these various data sources and our collection methodology (see [3] for details).

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.

Finding Public Vulnerabilities with the NVD

The NVD contains entries for all publicly released vulnerabilities assigned a CVE identifier, and rich annotations about the vulnerabilities. In particular, it summarizes the vulnerability, links to relevant external references (such as security advisories and reports), specifies the affected software, identifies the class of security weakness under the Common Weakness Enumeration (CWE) classifications, and evaluates the vulnerability severity using the Common Vulnerability Scoring System (CVSS).

We focused on the NVD as it is public, expansive, manually curated, and detailed. For this study, we analyzed a snapshot of the NVD taken on December 25, 2016. Its 80,741 CVE vulnerabilities served as our starting point for further data collection.

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 addressed the security vulnerability. We focused on popular Git web interfaces as they were the most commonly occurring (and Git overall is popular). Crawling these links afforded us the ability to collect security patches and access the source code repositories.

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. By manually investigating 100 randomly sampled commits, we found that all commits reflect fixes for the corresponding vulnerabilities, indicating the vast majority, if not all, of our commits are security patches. This data set corresponds to a variety of vulnerability types and severities, affecting an expansive range of products (from OS distributions to applications to libraries), detailed in [3].

Identifying General Bug Fixes

We can gain insights into any especially distinct characteristics of security patches by comparing them to bug fixes *in general*. However, to do so at scale we must automatically identify 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. We discuss the details of our classifier training and evaluation in [3].

With our classifier, we collected a data set of bug fixes by randomly selecting per repository up to 10 commits classified as bug fixes. This provided us with a large set of over 6,000 bug fixes (similar to our number of security fixes) balanced across repositories.

Processing Commits

In a patch, it can be useful to consider only changes to functional source code, rather than documentation files or source code comments. For each commit that we collected (both security and general bug fixes), we processed the commit data to produce an alternative "cleaned" version that filtered non-source code files and removed comments.

Estimating Vulnerability Public Disclosure Dates

Determining the public disclosure date of a vulnerability is vital to understanding the timeline of its life cycle. The CVE publication date indicates when the CVE entry was published, not necessarily when the vulnerability was publicly disclosed. To estimate the public disclosure date, we analyzed the external references associated with CVEs. These web pages frequently contain publication dates for information pertaining to vulnerabilities. Example pages include security advisories, public mailing list archives, other vulnerability database entries, and bug reports. We chose the earliest date among the extracted dates and the CVE publication date as our estimate.

Analysis Results

Our collected data set provides 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 general bug fixes. In this section, we discuss our more salient analyses and findings (see [3] for additional analyses).

We first consider the patch development process by connecting the vulnerability information available in the NVD with the historical logs available in Git repositories. We follow that by analyzing our collection of security and general bug fixes to help illuminate their differences, considering facets such as the complexity of fixes and the locality of changes. In general, to assess whether differences observed have statistical significance, we use permutation tests with a significance threshold of $\alpha = 0.05$ (discussed in detail in [3]).

Vulnerability Life Spans in Code Bases

Upon a vulnerability's discovery, we might naturally ask how long it plagued a code base before a developer rectified the issue, a duration we call the *code base life span*. Automatically and reliably determining this life span is difficult, requiring semantic understanding of the source code and 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. We assessed that this is a robust approximation through manual inspection of a random sample of security patches.

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 retrieved the last time each line was previously updated. We conservatively designate the most recent change date across all of the lines as the estimated vulnerability birth. The duration between this date and the patch commit date provides a lower bound on the vulnerability's code base life span. We observe that vulnerabilities exist in code bases for extensive durations, with a median life span of 438 days (14.4 months). Furthermore, a third of all CVEs had life spans beyond three years. The longest surviving vulnerability was a *21-year-old* information disclosure vulnerability in Kerberos.

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 bases through unanticipated public announcements have to quickly react before 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 may not release the available patch 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 data set with public disclosure dates (estimated as described in "Data Collection Methodology," above).

How frequently are vulnerabilities unpatched when disclosed? We observe that 21% of all vulnerabilities were not fixed at the time of public disclosure. 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, a quarter (26%) of these unpatched security issues remained unaddressed 30 days after disclosure, leaving a window wide open for attacker exploitation.

For the remaining 79% of all CVEs, project developers committed the security fixes by public disclosure time. 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? The degree to which security commits precede disclosures varies widely. This behavior highlights the security impact of an interesting aspect of the open-source ecosystem. 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 remedied in the project repository, it is unlikely to be widely fixed in the wild before public disclosure and update distribution. We note that over 50% of CVEs were patched more than two weeks before public disclosure, giving attackers ample time to develop and deploy exploits.

Patch Reliability

The patch that 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 identified instances of multiple commits for the same CVE, and classified the causes.

To locate CVEs associated with multiple commits where a subsequent commit may fix a previous one, we found CVEs listed in the NVD with multiple commits. Additionally, we attempted to identify further commits potentially associated with a CVE using repository Git logs, looking for commit messages that explicitly reference the original patch's commit hash or the CVE ID. Note that with this approach, we could only identify multiple patches when commit messages contained this explicit linkage, so our analysis provides a lower bound.

Filtering out duplicate commits (e.g., merges, rebases, and cherry-picks) as well as CVEs where all commits were within a 24-hour time window (thus even if there was a problem, it was quickly resolved), we found 440 CVEs with multiple commits.

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

Table 1: Summary of our manual investigation into 50 randomly sampled CVEs with multiple commits. Note that a CVE may have commits in multiple categories.

We randomly sampled 50 of the remaining 440 CVEs and manually investigated whether the fixes were problematic. Table 1 summarizes our results. We identified 26 (52%) as having incomplete fixes, 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. Other follow-on commits were benign, such as commits for documentation, testing, or refactoring. Note that some CVEs had multiple commits in multiple categories, resulting in the sum of CVEs in each category exceeding 100%. Problematic initial patches were followed by a median of one additional commit, with a median of 181.5 days and 33 days between commits for incomplete and regressive patches, respectively.

This random sample is representative of the 440 CVEs with multiple commits accounting for 14.2% of all CVEs. Extrapolating from the sample to 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 with unfortunate frequency, 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 data set might not have had enough time yet to manifest as ultimately requiring multiple commits.

Patch Complexity

How complex are security patches compared to bug fixes in general? Given the number and diversity of software projects we consider, we chose lines of code (LOC) as a simple-albeit-rudimentary metric.

Are security patches smaller than general bug fixes? Under the LOC metric, security commits overall are statistically significantly smaller than bug patches in general ($p \approx 0$). The median security commit diff involved 7 LOC compared to 16 LOC for general bug fixes. Approximately 20% of general bug patches had diffs with over 100 lines changed, while this occurred in only 6% of security commits.

Do security patches make fewer "logical" changes than general 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, we consider several lines updated as a single logical update, and a chunk of deleted code counts as a single logical delete. 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 general bug-fix commits. Between 30% and 40% of all commits for both security and general bug-fix commits also did not add any new code portions, indicating the majority of logical changes were updates to existing code.

Do security patches change code base sizes less than general 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. We observe that significantly more general bug patches result in a net reduction in project LOC, compared to security fixes: 18% of general 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, since the majority of all commits added to the code base. However, security commits tend to contribute less growth compared to general bug fixes, an observation that accords with our earlier results.

Patch 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 general bug fixes? We observe that security patches modify fewer files compared to bug fixes in general, a statistically significant observation ($p \approx 0$). In aggregate, 70% of security patches affected one file, while 55% of general bug patches were equivalently localized. Fixes typically updated, rather than created or deleted, files (mirroring code changes, which were typically updates). Only 4% of security fixes created new files vs. 13% of general bug fixes, and only 0.5% of security patches deleted files vs. 4% of general bug fixes.

Do security patches affect fewer functions than general bug fixes? We find that 5% of general 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.

Aspect of Security Patches	Summary of Results
Vulnerability Life Spans	Vulnerabilities often lived for years, with a third for more than three years.
Security Fix Timeliness	A fifth of vulnerabilities were not fixed at public disclosure time. When fixed before disclosure, the patches were visible in repositories weeks to months in advance.
Patch Reliability	We conservatively estimate that about 7% of security patches were incomplete and 5% regressive.
Patch Complexity	Security patches were significantly smaller than bug fixes in general.
Patch Locality	Security patches were more localized in their changes than general bug fixes.

Table 2: Summary of main analysis results.

Moving Forward

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 (summarized in Table 2) 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. 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. 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. In addition, software developers can already make strides in improving their testing processes by using existing tools such as sanitizers or fuzzers more extensively.

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 efforts, such as those supported by the Core Infrastructure Initiative [1], have already demonstrated that they can significantly improve the security of open-source software. 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.

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. From the public visibility into these commits, attackers can identify and weaponize the underlying vulnerabilities. The issue of vulnerability disclosure and embargoing of information is a complex debate, but the visibility of the patch itself should be part of that discussion.

Opportunities for leveraging characteristics of security patches. Our comparison of security patches with general bug fixes revealed that security fixes have a smaller impact on code bases, across 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, such as patch safety analysis. 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 [8] 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 vulnerabilities still heavily depends on patching, while the attack landscape has grown ever more dangerous.

Acknowledgements
This work was supported in part by the National Science Foundation awards CNS-1237265 and CNS-1518921.

References

[1] Core Infrastructure Initiative: https://www.coreinfrastructure.org.

[2] Z. Huang, M. D'Angelo, D. Miyani, and D. Lie, "Talos: Neutralizing Vulnerabilities with Security Workarounds for Rapid Response," in *Proceedings of the 37th I-EEE Symposium on Security and Privacy (S&P '16)*: https://www.eecg.toronto.edu/~lie/papers/zhuang_talos_oakland2016.pdf.

[3] F. Li and V. Paxson, "A Large-Scale Empirical Study of Security Patches," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS '17)*: https://www.icir.org/vern/papers/patch-study.ccs17.pdf.

[4] A. Ozment and S. E. Schechter, "Milk or Wine: Does Software Security Improve with Age?" in *Proceedings of the 15th USENIX Security Symposium (Security '06)*: https://www.usenix.org/legacy/event/sec06/tech/full_papers/ozment/ozment.pdf.

[5] J. Park, M. Kim, B. Ray, and D. Bae, "An Empirical Study of Supplementary Bug Fixes," in *Proceedings of the 9th Working Conference on Mining Software Repositories (MSR '12)*: https://web.cs.ucla.edu/~miryung/Publications/msr2012-supplementarypatch.pdf.

[6] US National Institute of Standards and Technology, National Vulnerability Database: https://nvd.nist.gov/home.cfm.

[7] S. Zaman, B. Adams, and A. E. Hassan, "Security Versus Performance Bugs: A Case Study on Firefox," in *Proceedings of the 8th Working Conference on Mining Software Repositories (MSR '11)*: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.740.4377&rep=rep1&type=pdf.

[8] H. Zhong and Z. Su, "An Empirical Study on Real Bug Fixes," in *Proceedings of the 37th International Conference on Software Engineering (ICSE '15)*: https://web.cs.ucdavis.edu/~su/publications/icse15-bugstudy.pdf.