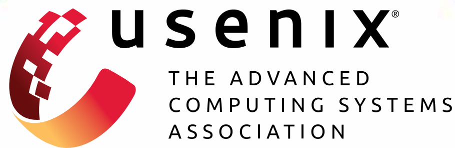


How Long Do Vulnerabilities Live in the Code? A Large-Scale Empirical Measurement Study on FOSS Vulnerability Lifetimes

Nikolaos Alexopoulos, Manuel Brack, Jan Philipp Wagner, Tim Grube, and Max Mühlhäuser, *Technical University of Darmstadt*

https://www.usenix.org/conference/usenixsecurity22/presentation/alexopoulos

This paper is included in the Proceedings of the 31st USENIX Security Symposium.

August 10–12, 2022 • Boston, MA, USA

978-1-939133-31-1

Open access to the Proceedings of the 31st USENIX Security Symposium is sponsored by USENIX.

How Long Do Vulnerabilities Live in the Code? A Large-Scale Empirical Measurement Study on FOSS Vulnerability Lifetimes

Nikolaos Alexopoulos, Manuel Brack, Jan Philipp Wagner, Tim Grube and Max Mühlhäuser
Telecooperation Lab, Technical University of Darmstadt, Germany

Abstract

How long do vulnerabilities live in the repositories of large, evolving projects? Although the question has been identified as an interesting problem by the software community in online forums, it has not been investigated yet in adequate depth and scale, since the process of identifying the exact point in time when a vulnerability was introduced is particularly cumbersome. In this paper, we provide an automatic approach for accurately estimating how long vulnerabilities remain in the code (their *lifetimes*). Our method relies on the observation that while it is difficult to pinpoint the exact point of introduction for one vulnerability, it is possible to accurately estimate the average lifetime of a large enough sample of vulnerabilities, via a heuristic approach.

With our approach, we perform the first large-scale measurement of Free and Open Source Software vulnerability lifetimes, going beyond approaches estimating lower bounds prevalent in previous research. We find that the average lifetime of a vulnerability is around 4 years, varying significantly between projects (~2 years for Chromium, ~7 years for OpenSSL). The distribution of lifetimes can be approximately described by an exponential distribution. There are no statistically significant differences between the lifetimes of different vulnerability types when considering specific projects. Vulnerabilities are getting older, as the average lifetime of fixed vulnerabilities in a given year increases over time, influenced by the overall increase of code age. However, they live less than non-vulnerable code, with an increasing spread over time for some projects, suggesting a notion of maturity that can be considered an indicator of quality. While the introduction of fuzzers does not significantly reduce the lifetimes of memory-related vulnerabilities, further research is needed to better understand and quantify the impact of fuzzers and other tools on vulnerability lifetimes and on the security of codebases.

1 Introduction

Software flaws that can potentially be exploited by an adversary are referred to as *security bugs* or *vulnerabilities*. Reducing the number of vulnerabilities in software by finding existing ones and avoiding the introduction of new ones (e.g. by employing secure coding practices or formal verification techniques) is one of the primary pursuits of computer security.

Measurement studies on the different stages of the *vulnerability lifecycle* play an important role in this pursuit, as they help us better understand the impact of security efforts and improve software security practices and workflows. The community has produced a number of such outputs in recent years [5, 8, 14, 18, 23, 28, 36]. A vulnerability's lifecycle, or *window of exposure* as introduced by Schneier [34], describes the phases between the introduction of a vulnerability in the code, and the point in time when all systems affected by that vulnerability have been patched. There have been several adaptations of the vulnerability lifecycle concept (e.g. w.r.t. the number of phases or their ordering and its non-linearity), but the general concept remains the same, and a simple version is shown in Fig 1. The lifecycle of a vulnerability (or

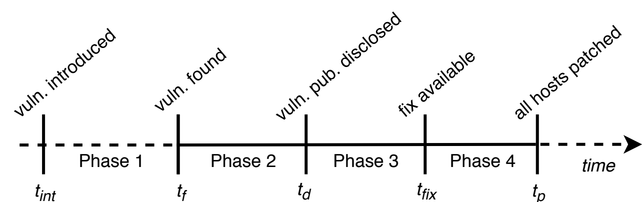


Figure 1: Simplified plot of a vulnerability's lifecycle. Continuous line shows period of possible exploitation.

alternatively the window of exposure to a vulnerability) begins with its introduction into a product (at time t_{int}). This first phase (Phase 1) of its lifecycle ends with its discovery by some party (t_f). Phase 2 covers the time period during which a vulnerability is known to at least one individual (and there is an associated risk of exploitation depending on their intentions), but is not publicly disclosed yet. Phase 3 begins with the public disclosure of the vulnerability (t_d) and ends

with the publication of a patch fixing the vulnerability (t_{fix}). Finally, Phase 4 ends when all vulnerable hosts have been patched (t_p). The phases described above can be long, short or even non-existent, depending on the specific vulnerability and the processes of the affected product's vendor. For example, if a vulnerability is discovered by an ethical hacker and responsibly disclosed, the software vendor has the opportunity to eliminate Phase 3 by disclosing the vulnerability together with the fix. This is common practice for many projects. In some cases (more often for proprietary software) a vulnerability can be silently patched, meaning public disclosure never occurs, and the phases may differ significantly from the figure.

While the latter phases of the vulnerability lifecycle have received renewed research attention [14, 23, 36], this has not been the case for the early phases. A particularly interesting quantity describing the early part of the lifecycle (Phases 1–3), is the amount of time a vulnerability remains in the (upstream) codebase of a project. In the context of a version control system, it is the time between a Vulnerability Contributing Commit (VCC), and a fixing commit. We refer to this quantity, as a vulnerability's *code lifetime*, or just *lifetime* for the rest of this paper. This quantity can provide valuable insights regarding code maturity, can guide practical decisions, and can help us investigate fundamental questions: *Is the quality of software improving? How long should the "stable" freeze last? Are some vulnerabilities harder to find than others? What impact do automated testing tools have on the lifetimes of vulnerabilities?*

Previous approaches towards measuring vulnerability lifetimes either relied on manual mappings of fixing commits to VCCs [10, 11, 28], which meant they were limited in scale (low number of vulnerabilities affecting one project), or used heuristics to estimate lower bounds [18]. Li and Paxson [18], in particular, provided lower bound estimates for vulnerability lifetimes using an automated approach, as part of their large scale study on security patches.

Research Questions. Although accurately estimating the lifetime of vulnerabilities is a useful contribution on its own, the real power of a metric comes from the insights derived from its application. Vulnerability lifetime, as a metric, is less affected by the bias of vulnerability-hunting scrutiny, compared to metrics based on counting the raw number of discovered vulnerabilities[1]. Therefore, it can provide valuable novel insights. The main research questions we set off to answer in this paper are summarized by the following points:

– *How long do vulnerabilities remain in the code? Do these lifetimes differ for different projects or different vulnerability types? How long does it take to find certain portions of ultimately discovered vulnerabilities? (e.g. 25/50/75 percent)* Answering these questions will provide us with insights regarding the duration of the window of exposure for different projects. Results can aid decisions regarding the amount of time a "stable freeze" (only critical patches are applied to the software) should last, and for how long investing on dedicated long-term security support for a stable version may be necessary.

– *Are lifetimes increasing or decreasing over time? Are there signs of improved quality?* Answering these questions will help us approach a fundamental question of software security from a novel angle: *is software getting more secure over time?*

Contributions. We provide the first approach for accurately estimating vulnerability lifetimes automatically, going beyond estimating lower bounds, which is prevalent in previous work. The approach is based on a heuristic that receives as input a CVE's set of fixing commits and produces as output an estimate of the CVE's lifetime, utilizing information available in a project's version control system. We rigorously validate our approach on a dataset of 1,171 ground truth mappings between CVEs and their VCCs. Then, we perform lifetime measurements on a large dataset of 5,914 CVEs, spanning 11 popular Free and Open Source Software (FOSS) projects (selection criteria for the projects are presented in Section 3.2). This dataset is, to the best of our knowledge, the largest and most complete dataset of mappings between CVEs and their fixing commits, in existence.

Main findings. Vulnerabilities generally remain in the code for large periods of time, varying significantly between projects. An exponential distribution is a good fit for vulnerability lifetimes overall, and for individual projects. Overall, vulnerability lifetimes increase with time, however the shape of their distribution remains exponential. A vulnerability's lifetime does not depend on its type. Lifetimes are closely correlated with the general code age of a repository. However, vulnerable code lives less than non-vulnerable code. This spread (between the age of vulnerable and all code) increases over time for some projects.

2 Related work and background

In this section, we place related work in context and define vulnerability lifetime in version control systems.

2.1 Related work on vulnerability measurements

There exists a considerable amount of work on measuring different aspects of software security. These range from general studies on bug characteristics [37], to studies on the vulnerability discovery rate and its trends in various software projects [1, 8, 33]. Another strand of work focuses on vulnerability discovery models that try to capture the vulnerability discovery rate of specific software products after their release. Most of these models try to model the after-release discovery rate as a function of time [2–4, 15, 16], and others as a func-

[1]The futility of software quality arguments based on the raw numbers of vulnerability discoveries is well-documented [7, 17].

tion of expended effort, either measured as the market share of a specific product [3], or the cumulative user months estimated to have elapsed since its release [40]. These reliability-inspired discovery models most often focus on a specific version of software (static codebase) and their accuracy against empirical data has been contested [1, 26, 27]. Specifically, empirical evidence [1, 33] suggest that there might be no decreasing trend in the rate of vulnerability discoveries, or if there is, it may be attributed to a decrease in the detection effort, rather than the depletion of vulnerabilities. Therefore, the community has identified the need to measure different aspects of the security process.

There are studies that measure the duration of different parts of the vulnerability lifecycle (see Figure 1). Frei [14], Krebs [17] and Shahzad et al. [36] measure characteristics of Phase 3 of the vulnerability lifecycle, namely how fast patches are developed and made available, and what the impact of delays is in terms of real-world exploits. Phase 4 of the lifecycle, namely how fast patches are applied to vulnerable systems, and how this relates to attacks in the wild, is studied by Nappa et al. [23]. Phase 2 of the lifecycle, namely how long vulnerabilities remain undetected since they have been discovered by attackers and used in zero-day attacks, is studied by Bilge et al. [5].

The main topic of this paper, *vulnerability lifetimes*, i.e. the amount of time vulnerabilities remain in the code in the (upstream) repositories of non-static projects, has received attention, but limited progress has been made to date. A reason may be the difficulty of automatically assessing when a vulnerability was introduced in a codebase. In a short LWN.net article, Corbet [11] presented the results of a small-scale study on 80 CVEs affecting the Linux kernel. This was followed by a blog post from Cook [10] on the amount of time 557 Linux kernel CVEs remained in the code. Both of these studies relied on manually curated data (either of the author or from the Ubuntu Security Team) and only touched the surface of the problem, being limited to providing one plot of the data. However, the interest they raised, expressed in lengthy discussions in their respective forums, acted as a motivation for our work. The only study that investigates the issue of vulnerability lifetimes in a technical paper is, to the best of our knowledge, the recent work by Li and Paxson [18]. A small part of their insightful large-scale study on security patches in open source software is dedicated to assessing a lower bound for vulnerability lifetimes. Using an approximation of the exact value rather than a lower-bound approach, our results regarding vulnerability lifetimes differ by an order of magnitude compared to theirs. Also, due to more detailed per-project analysis, our conclusions do not support their hypothesis that vulnerability lifetimes and their types are correlated. In our study, we focus on how lifetimes differ between different projects and how they develop over the years, questions that generated interesting insights and have never been rigorously investigated before.

2.2 Vulnerability lifetimes in version control systems

A vulnerability's *code lifetime*, or just *lifetime* for the rest of this paper, has been informally defined as *the amount of time a vulnerability remains in the codebase*. This is the time that elapses between a change in the codebase that introduced a weakness[2], and the change in the codebase that fixed the weakness (which was discovered in the meantime). In a version control system, such as git, SVN, or mercurial, these changes are part of commits. These commits include metadata about each change (e.g. commit timestamp, author) and the complete history of a repository can be tracked via a tree-like structure of commits (including branching and merging commits). A commit that contributed to the introduction of a weakness is known in literature [20, 31] as a *Vulnerability Contributing Commit (VCC)*, and a commit that helped resolve the issue is referred to as a *fixing* commit. A vulnerability may have multiple VCCs and fixing commits due to several reasons. Some examples follow:

– a CVE may cover multiple programming errors. For example, CVE-2019-10207 describes a flaw in the bluetooth drivers of the Linux kernel. The fixing commit[3] indicates that checks were missing in the bluetooth driver files of several manufacturers, pinpointing 5 responsible VCCs with commit dates spanning 8 years. Another example is CVE-2015-8550, which describes flaws in 2 linux kernel virtualization drivers (Xen blktap and Xen PVSCSI), introduced in 2013 and 2014 respectively, and fixed with a patch spread among 7 commits all with the same commit date in 2015[4].

– a vulnerability may be removed and then re-introduced. For example CVE-2017-18174 describes a double free in the Linux kernel that was introduced in 2015, removed without being designated as a security issue roughly a year later as part of a commit that "cleaned the error path", re-introduced by a commit that provided new functionality 6 months later, and finally fixed again within a month of re-introduction[5].

– a fix may be extensive, requiring multiple commits. For example the fix of CVE-2017-9059 included considerable refactoring of the code and changes spanned 2 fixing commits

[2] What constitutes a weakness is open to definition and often also to discussion among project contributors/developers. In the empirical part of this paper, we count vulnerabilities by their CVE identifier in the NVD, as often done in literature. We note that the CVE identifier is a level of abstraction higher than individual weaknesses, in the sense that multiple related weaknesses may be grouped together under a unique CVE identifier. However, for the sake of text simplicity, for the rest of the paper, we do no differentiate between the concepts (1 CVE-ID = 1 vulnerability).

[3] https://github.com/torvalds/linux/commit/b36a1552d7319bbfd5cf7f08726c23c5c66d4f73

[4] more information at https://bugs.launchpad.net/ubuntu/+source/linux/+bug/1530403.

[5] more information at https://bugzilla.redhat.com/show_bug.cgi?id=1544482.

committed within a day[6]. Another Linux kernel vulnerability with CVE-2012-2119, this time a buffer overflow, had a patch spanning 5 short fixing commits modifying the same file, all committed on the same day[7].

Defining a lifetime metric comes down to explicitly defining the start and end points of the time measurement.

- **Which of the potentially multiple VCCs/fixing commits should we consider as a start/end point?** Considering the causes for multiple commits either introducing or fixing a vulnerability, we decide to use the first of the VCCs as the start of the time measurement and the last of the fixing commits as the end. For cases like the first example concerning multiple sub-vulnerabilities described in one CVE, we therefore measure the lifetime of the oldest one. For re-introduction cases, like the one in the second example, we measure the total lifetime. In those cases, this whole period is most often a period of risk for systems running "stable" software versions (only applying critical patches), since the premature fix is often not designated as a security issue, and therefore not considered a critical patch.
- **Which timestamp to use?** Most projects use a "main" branch of development (traditionally *master* in git) to track their code, and have a public mirror of this repository so that users can download the most recent versions of the code[8]. Changes are then developed, discussed, and tested in private copies of the repository. When a change is ready to make it to the main branch, it is either (a) prepared as a (typically short) series of commits based on a recent reference commit and then merged into the main branch, or (b) directly committed to the main branch. For the purpose of measuring the lifetime of a vulnerability as a part of its window of exposure, we would want to measure the time between the VCC and the fix being widely available to users (including, e.g. maintainers of software distributions). Following this argumentation, we would use the time a VCC or a fixing commit was merged into the main branch as its timestamp, rather than its "commit timestamp" (because the change may potentially be kept private until merging). This was indeed our first approach. However, from our empirical results we came to the conclusion that this adds unnecessary complexity to both the definition and the computation of the metric. The time between the commit and its merging into the main branch is usually very short (average of $\sim$20 days) in comparison to the lifetime of a vulnerability (average in years). This is due to the fact that the usual practice after a fix is prepared and tested, is to download the most recent version of the code from the main branch of development and create a patch against this version. Furthermore, more complexity in the definition would have been needed to account for fixes that were never merged into the main branch because the vulnerability they addressed only affected older versions of the software (and were therefore merged into e.g. a stable branch). Considering the above, we backtracked and decided to use the commit timestamp, which is readily available in the metadata of a commit in all three popular version control systems (git, SVN, mercurial).

[6] more information at https://www.spinics.net/lists/linux-nfs/msg63334.html.

[7] more information at https://bugs.launchpad.net/ubuntu/+source/linux/+bug/987566.

[8] see https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git for Linux.

3 Dataset creation

In this section we describe the data collection and cleaning process. This process allowed us to create the largest and most complete, to the best of our knowledge, datasets of (a) mappings between a CVE and its VCC(s), and (b) mappings between a CVE and its fixing commit(s). We therefore consider the dataset to be a valuable contribution in itself and will make it publicly available for other researchers to use[9].

3.1 Linking CVEs to their VCCs

An integral part of our dataset are mappings from CVEs to their VCCs to create a "ground truth" dataset. These mappings come from manually curated datasets of researchers and project maintainers and enable us to evaluate and validate methods that automatically estimate lifetimes of vulnerabilities for which no such ground truth data are available.

The largest source we identified for such mappings, is the Ubuntu CVE Tracker [6]. In this project, the Ubuntu Security Team gathers and curates several data points on vulnerabilities affecting the Linux kernel, often including their fixing commits and VCCs. Note that we exclude some of the mappings in the project from our dataset as their corresponding VCC refers to the initial git commit of the Linux kernel, leaving us with 1,202 mappings between 885 CVEs and their VCCs. Our reasoning is that vulnerabilities found in this commit were introduced before the beginning of the git era and thus using the timestamp of the initial git commit would let their lifetime appear shorter than it actually is.

Additionally, we found 436 ground truth mappings for Chromium[10] (226 CVEs), and 163 for the Apache HTTP Server (httpd) (60 CVEs) in the Vulnerability History Project [21]. The project also contains a few additional data points for repositories, other than Chromium and httpd, however these repositories do not use C/C++ as their main programming language. Since the quantity of those mappings for other programming languages was too low to validate the accuracy of our methodology on them, we decided to limit our dataset to C/C++ repositories. Overall we were able to map 1,171 CVEs to one or more VCCs, from high-quality sources.

[9] dataset available at https://figshare.com/s/4dd1130c336f43f6e18c

[10] Chromium is a web browser on its own right, but its codebase is also used as a basis for Google Chrome, as well as other web browsers.

3.2 Included projects

For our analysis, we want a large representative sample of FOSS projects that are compatible with our ground truth datasets and provide enough data points for our analysis. As starting point, we established the following requirements for software projects to be included:

(a) The project should be free and/or open source with transparent and consistent security workflows. In fact, all projects that were included at the end of this process are distributed under Debian[11] as free software. Nonetheless, they are widely used in other operating systems and make up a good representation of FOSS in general.

(b) The project should have a considerable number of reported CVEs. In order to allow a thorough analysis of all projects, we limited ourselves to those with at least 100 CVEs to ensure meaningful results (we expect statistical arguments to be possible). This condition forced us to discard many projects we considered interesting, simply because they did not have enough data available in the National Vulnerability Database (NVD).

(c) The project should be mainly written in C or C++. Intuition suggests that our methodology would provide results of similar quality for other programming languages with similar syntax and semantics, such as Java. However, we did not have enough ground truth data available to empirically prove this hypothesis for other programming languages, and therefore focus on C/C++.

(d) Lastly, we require a significant number of a project's CVEs to be linkable to a fixing commit. We had to discard some projects because it was not possible to consistently identify the fixing commit(s) for the project's CVEs.

In the end we created a dataset consisting of 11 different projects of various sizes, ages and areas of application. All included projects and their respective numbers of CVEs and corresponding fixing commits are listed in Table 1. We thoroughly investigated commit messages, bug tracking systems and NVD references to ensure the highest possible – to the best of our efforts – yield of vulnerability mappings. A general description of the process follows in the next section.

3.3 Linking CVEs to their fixing commits

Our data collection process is based on information from the NVD [25]. The NVD is manually curated and represents one of the largest collections of software vulnerabilities. Consequently, it is frequently used in research on software security, having generated valuable insights on various topics. At the same time, the NVD has some well-known pitfalls, that the process described below tries to mitigate. This issue is further discussed in Section 7. For our work, we relied on the CVE-search tool [13] to obtain a local copy of the NVD for querying and obtaining data.

[11] https://www.debian.org/

For our analysis of vulnerability lifetimes, we needed to link a vulnerability entry in the NVD to one or multiple *fixing commits* resolving the underlying flaw. In order to create as large a dataset as we could for our evaluation, we applied and combined four different approaches.

1. CVE-ID in Commit Message. Some fixing commits mention the related CVE-ID in the commit message, establishing a link between CVE and fixing commit. We investigated all these mappings using a combination of automated scripts and manual effort. First, we deemed correct all mappings that mention the CVE-ID in a project-specific syntax (e.g Bug-Id: CVE-2017-9992) that clearly denotes a matching CVE. We manually analyzed the remaining 176 mappings, and removed 17 unjustified mappings, corrected 2, and added 2[12].

2. Commit in NVD Reference. Entries in the NVD often contain references to third party websites that include security advisories, bug tracking systems, and also links to commits in a project's version control system. In the cases where a reference to a commit in the respective repository was included, we considered this commit to be a fixing commit. We manually validated the mappings obtained using this method by investigating a random subset of 50 samples. We found that all of the sampled commits were indeed fixes for the respective CVEs.

3. Common Bug ID. Many software vendors use dedicated bug tracking systems in their development process. Each commit corresponds to a bug identifier that is denoted in the commit message using a particular syntax. Thus, most commits can be linked to a certain bug ID. NVD entries, on the other hand, may contain a link to the vendor's online bug tracking system in their references. Using regular expressions, the bug ID can be extracted from such links and then matched to corresponding commits. We assume that the developers mentioned the correct bug identifiers in their commit messages and the curators of the NVD reference the correct bugs as well. We recognize that this is a potential threat to the correctness of these mappings (see Section 7), but are confident the errors are negligible.

4. Third party mappings. As the information provided by linking vulnerabilities and fixing commits is useful for various purposes, fellow researchers and security experts have collected similar data for some software projects. In addition to our own analysis, we incorporated this data, especially in cases where acquiring the information in an automated fashion proved particularly difficult. We relied on three different third party mappings: (a) data provided by the *Linux Kernel CVEs* project [19], (b) mappings that were manually curated by Piantadosi et al. [32], (c) information from the Debian Security Tracker [12]. We are confident in the quality of data obtained from these parties, as they are either curated from reliable sources or relied on a data collection methodology similar to ours.

[12] There were two occasions where a fix belonged to multiple CVEs as indicated in the commit message

These efforts resulted in a dataset of 5,914 CVEs across 11 projects that can be linked to one or more fixing commits. The corresponding numbers of CVEs as well as the resulting number of mappings from this process are listed in Table 1. CVEs with associated fixing commits are a subset of all of the CVEs in the NVD. For the calculation of vulnerability lifetime we could only consider those CVEs, where an associated fixing commit could be identified in the code repository. Rather than being a limitation of the approach, this addresses an important problem with the NVD regarding the reliability of information on vulnerable products and versions (highlighted in previous research [23,24]). Contrary to Li and Paxson [18]

Project	**CVEs**	**w/ fix. com.**	**# fix. com.**
Linux (kernel)	4,302	1,473	1,528
Firefox	2,179	1,498	3,751
Chromium	2,781	1, 580	2,820
Wireshark	600	314	343
Php	663	281	932
Ffmpeg	326	277	373
Openssl	214	144	259
Httpd	248	132	476
Tcpdump	167	115	128
Qemu	340	213	290
Postgres	139	76	141
Total	**11,959**	**5,914**	**11,041**

Table 1: Number of CVEs and mappings per project. First column gives the total number of CVEs returned from a search of the NVD. Second column gives the number of those CVEs for which at least one fixing commit was found in the project repository. Third column gives the total number of fixing commits found per project.

we focused our efforts on a smaller set of projects with a significant number of NVD entries available. We dedicated considerable efforts towards achieving the highest mapping rate that we could for these projects. In order to do that we made sure to identify all potential syntaxes for denoting bug IDs, all websites and corresponding hyperlinks belonging to the same bug tracking system as well as combining multiple of the introduced techniques. Consequently, we believe our dataset to be the most complete mapping for the included projects. Additionally, we also consider it to be more diverse than the one used in Li and Paxson's work, as over 40% of their data originates from the Linux kernel or an operating system based on it.

4 Lifetime estimation

In this section, we describe our methodology for automated lifetime estimation from vulnerability fixing commits. We start off by evaluating a lower-bound approach used in previous work [18]. To the best of our knowledge, we are the first to evaluate this approach against ground truth data.

4.1 Lifetime estimation in previous work

Calculating the lifetime of a vulnerability is a non-trivial task, as finding VCCs can be very difficult [20]. One approach used before is manually identifying VCCs via thorough analysis of the fixing commits of a vulnerability [28]. While this method likely yields the most accurate results, it is unsuitable for large scale studies. Therefore, Li and Paxson [18] opted for an automated approach to the problem. They approximated a lower bound of a vulnerability's lifetime by using the *git blame* command. *Git blame* finds which commit last changed a specific line in a given file, and so can be used to trace a vulnerability back to its origin. Li and Paxson's approach was to run the command on every deleted or modified line of a given fixing commit; this process usually returns multiple candidate VCCs. They then picked the most recent of the commit dates of those VCCs as their vulnerability introduction date, going for a lower bound approach.

Although our empirical evaluation showed that this approach is indeed correct for getting a lower bound on the lifetime of a vulnerability, we found it too conservative for our needs. To be precise, we found that it underestimates the average lifetime for vulnerabilities in our ground truth dataset by 346.88 days. Due to the large underestimation of average lifetime, we deem this approach unsuitable for our study.

A heuristic similar to Li and Paxson's was used by Perl et al. in VCCFinder [31], albeit for a different goal. Their *git blame*-based heuristic aims to pinpoint the exact VCC, with the goal of creating a dataset suitable for training a classifier that can flag risky commits. Contrary to the Li and Paxson approach (that only blames lines that were deleted or modified in the fixing commit), the VCCFinder heuristic also takes into account lines added by the fixing commit. The commit that is blamed the most often is then marked as the VCC.

In VCCFinder, the accuracy of the heuristic is calculated at 96%. This figure was derived by the authors by taking a 15% sample of VCCs (96 in total) that the heuristic identified, and manually checking them. Naturally, simply applying this heuristic seemed like a good fit for our use-case. Unfortunately, in our evaluation of the heuristic against ground truth data, we could not observe similar accuracy (accuracy ~40%). We attribute the optimistic evaluation of the heuristic by Perl et al. to the difficulty of pinpointing VCCs manually. Our results seem sensible considering the fact that software developers put a significant amount of manual effort[13] into *regression tracking* which includes identifying VCCs. Thus, it seems likely that the accuracy of a relatively simple heuristic like this, is limited. However, for the purpose of training a classifier that pinpoints "risky" commits, with the aim to

[13] https://lore.kernel.org/lkml/3519198.TemPj1OATJ@vostro.rjw.lan/
https://yarchive.net/comp/linux/regression_tracking.html

drastically reduce the search space for subsequent manual auditing, such accuracy is perfectly acceptable. Indeed, the authors of VCCFinder showed that it is successful in finding previously unflagged vulnerabilities and outperformed existing approaches.

4.2 Our approach

A key observation is that we do **not** need to pinpoint the exact VCCs for our purposes. It is sufficient to approximate the point in time when a vulnerability was introduced. We found that we can estimate this the closest by repurposing the VCCFinder heuristic with some slight modifications (similar to the ones by Yang et al. [41]). We modify the heuristic in such a way that it returns an approximation (in days) of how long the code was vulnerable instead of the exact VCC. This is done by averaging multiple possible dates when the vulnerability could have been introduced, and assigning different weights corresponding to how often each individual commit was blamed. Our approach is as follows:

1) We use *git blame -w*[14] to map every "interesting" change of a fixing commit to potential VCCs:

– Ignore changes to tests, comments, empty lines and non C/C++ files.

– Blame every line that was removed.

– Blame before and after every added block of code (two or more lines) if it is not a function definition as these can be inserted arbitrarily.

– Blame before and after each single line the fixing commit added if it contains at least one of these keywords ("if", "else", "goto", "return", "sizeof", "break", "NULL") or is a function call. This approach was shown to perform well in blaming actual VCCs by Yang et al. [41].

2) We then calculate our estimated introduction date d_h from the list of blamed commits. For n commits we have:

$$d_h = d_{ref} + \frac{1}{\sum_{i=1}^{n} b_i} \sum_{i=1}^{n} b_i (d_i - d_{ref}) \qquad (1)$$

where b_i is determined by the number of blames the commit i received and d_i is the respective commit date. For easier calculation, the dates are represented as difference in days to a static reference date d_{ref} (January 1, 1900).

Using this heuristic, we decrease the mean error on our ground truth set by 66% compared to using the lower-bound blaming heuristic of Li & Paxson, effectively overestimating the actual average lifetime by 117 days (see Table 2). However, having a low error on the ground truth dataset does not necessarily mean that the approach is suitable for our needs. We have to address some additional important points regarding the robustness of the approach.

[14] All projects that we investigated either used git or offered a git mirror of their repositories. However, the approach is generalizable, as other version control systems offer similar commands (e.g. *svn blame* or *hg annotate*).

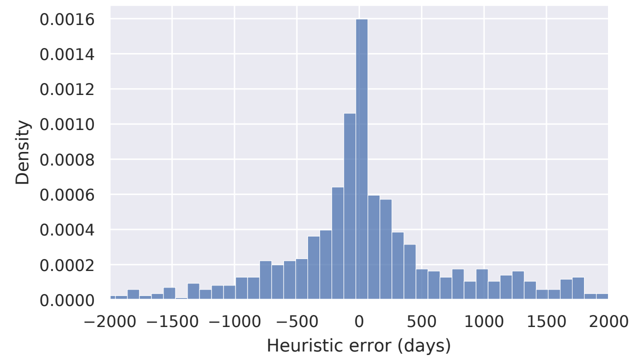


Figure 2: Distribution of heuristic errors in days (excluding data points with no error for readability). Equally sized bins.

How many data points are needed? A standard deviation of more than 900 days means that individual measurements are generally not reliable. Therefore, we need to rely on measurements of the mean over larger sample sizes. Treating the errors as independent random variables sampled from the error distribution of Figure 2, we can compute the standard deviation of the sample mean for a given sample size with the Bienaymé formula. According to the Central Limit Theorem, the distribution of the sample mean will be normal for a large enough number of samples. We empirically conclude that at least for 20 or more samples, this holds for the distribution of the errors. Therefore, we can compute bounds for the sample mean using normal distribution tables. For example, for a sample size of 20 and a confidence level of 95%, the sample mean will lie in the interval [117-395, 117+395] days. Making the simplification that the mean error is 0 (necessary for generalization), we can say that for a confidence level of 95%, the margin of error will be ± 395 days for 20 samples, ± 250 days for 50 samples, and ± 176 days for 100 samples (compared to an average lifetime of almost 1150 days, calculated from the ground truth data). We consider the margin of error for 20 samples as the maximum tolerable error for our study, and set the minimum number of data points for a measurement to 20. We, therefore study means of at least 20 CVE samples, although for most of our analysis (e.g. average lifetimes per project, vulnerability types), we consider means of 60 or more samples. Note that the calculations above are approximate and their aim is to get a lower bound for the number of samples required for meaningful estimation.

Does the weighted average heuristic generalize (over time and between projects)? Although we have a limited number of data points in our ground truth dataset, we can see that the performance of our heuristic is similar for the three different projects (see Table 2). Another source of confidence in the robustness of our heuristic is that its error is symmetrically (nearly normally) distributed around a low mean value (that can be assumed to be zero), as can be seen in Figure 2. In-

Project	CVEs	Li & Paxson		our approach (VCCFinder)		our approach (w. average)		Lifetime
		Mean error	St. dev	Mean error	St. dev	Mean error	St. dev	Mean
Linux (kernel)	885	-323.74	1,033.27	157.51	1,127.60	163.11	994.01	1,330.85
Chromium	226	-370.32	747.51	-15.54	754.19	-38.44	633.45	754.22
Httpd	60	-599.80	1,160.05	257.45	915.81	22.40	868.91	1,890.23
All CVEs	1,171	-346.88	993.72	129.24	1,057.9	117.00	932.52	1,248.22

Table 2: Comparison of heuristic performance for the lower-bound approach of Li&Paxson, our approach based on a repurposed version of the VCCFinder heuristic [31], and our optimized heuristic (weighted average). All against ground truth data and measured in days.

tuitively, this means that the errors "automatically" cancel out. For example, an alternative approach to have more accurate estimations of the lifetime would be to find a suitable "perfect constant" to add to the lower bound estimation as computed by the Li & Paxson approach. However, we found that depending on the data sampled, a perfect constant generally performs worse than our weighted average approach. Taking Chromium as an example, calculating a constant up to some date X and then adding it to the output of the lower bound estimation approach, does not provide a good estimate compared to the weighted average heuristic. For example, assume the available data for Chromium is split into two subsets with fixing commits before 2014, and with fixing commits in 2014 and after. Calculating a perfect constant on the first dataset and applying it to the second, results in an average underestimation of 228.58 days, while the weighted average heuristic underestimates the correct lifetime by 33.56 days. Also, the calculated constants between the Linux kernel and Chromium differ drastically (the kernel constant would be up to 6 times larger), which shows that this approach cannot be transferred between different projects, whereas our approach does not require tuning and provides good results for all three projects with ground truth data that are available.

Can we use the heuristic to assess trends and distributions? Figure 3 shows the (ground truth) lifetimes of Linux kernel[15] vulnerabilities in our ground truth dataset (867 CVEs), per year from 2011 to 2020. For the same CVEs, it also shows the estimate by our weighted average heuristic, in addition to the best linear fits for the trend in each case. Visual inspection of the plot as well as the relatively small difference in the calculated best linear fits (gradient of 163 days/year for heuristic, 155 days/year for ground truth) supports the assertion that the heuristic can be used to study lifetime trends over time.

To assess whether the heuristic is also suitable for estimating the distribution of vulnerability lifetimes, we plot and compare the histograms of the ground truth data and the output of the heuristic for the same CVEs (Figure 4). The histograms showcase similar characteristics and the exponential

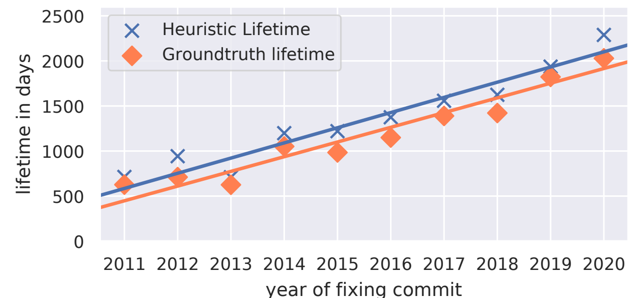


Figure 3: Comparison of heuristic vs ground truth values for historic trend of vulnerability lifetimes in the linux kernel. Linear OLS fit.

distribution appears to be a good fit to both the heuristic and ground truth data for most of the data range (see Q-Q plots in Figure 4). Overall, our experiments support the assertion that the heuristic can be used to study the distribution of lifetimes.

5 Results

In this section, we present the results of applying our weighted average heuristic for lifetime estimation on a large dataset of 5,914 CVEs with available fixing commits[16]. For 1,171 of those data points we use ground truth data, and for the rest we use the approach described in Section 4 to approximate their lifetimes.

5.1 General

Table 3 shows an overview of the computed lifetimes for each project, as well as for the dataset as a whole. In general, vulnerabilities live in the code for long periods of time (over 1,900 days on average). This fact has also been indicated by previous research [1, 10, 11, 28]. We observe large differences between projects (TCPDump has 4 times the average lifetime of Chromium). There can be multiple possible explanations for these differences (e.g. better security protocols, general

[15] We use the Linux kernel for this investigation, since it is the project with the most data points in our ground truth dataset (see Table 2). However, the same conclusion can be made by using the ground truth data for Chromium.

[16] Due to the restrictions of the heuristic (e.g. only consider changes in C/C++ files) the total number of CVEs with an estimated or ground truth lifetime is 5,436.

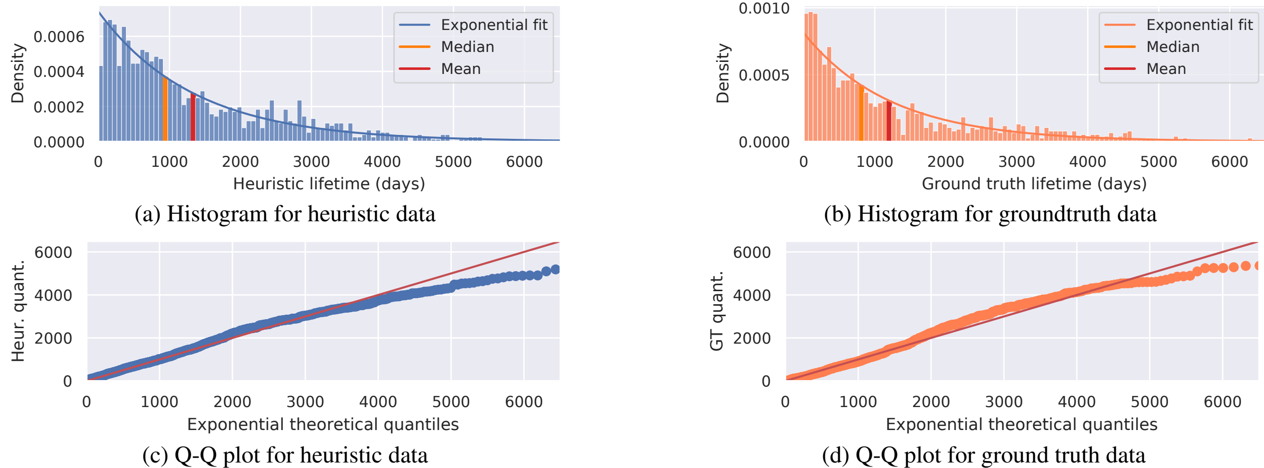


Figure 4: Comparison of histograms of lifetime distributions between heuristic and ground truth data for the same CVEs. The exponential fit to the histograms and the corresponding Q-Q plots are also provided.

development, code churn) that we further touch upon in the following subsections. Also, we observe that the median is generally lower than the mean. This gives an indication regarding the distribution of lifetimes within a project. This issue is specifically tackled in the following subsection.

Project	Lifetime Average	Lifetime Median
Linux (kernel)	1,732.97	1,363.5
Firefox	1,338.58	1,082.0
Chromium	757.59	584.5
Wireshark	1,833.86	1,475.0
Php	2,872.40	2,676.0
FFmpeg	1,091.99	845.5
OpenSSL	2,601.91	2,509.0
Httpd	1,899.96	1,575.5
Tcpdump	3,168.58	3,236.0
Qemu	1,743.86	1,554.0
Postgres	2,336.56	2,140.0
Average of projects	***1,943.48***	***1,731.0***
All CVEs	***1,501.47***	***1,078.0***

Table 3: Overview of average lifetimes per project (ordered by number of CVEs)

5.2 Distribution

Figure 5 shows the distribution of lifetimes for all CVEs, along with an exponential fit. Upon initial visual inspection of the histogram of Figure 5, we selected the exponential distribution as a potentially good fit to the data. Then, by employing a Q-Q plot [39] we verified that the exponential distribution is indeed an excellent fit for lifetimes of up to around 4,200 days (Figure 6), accounting for 94% of our data. Then, the

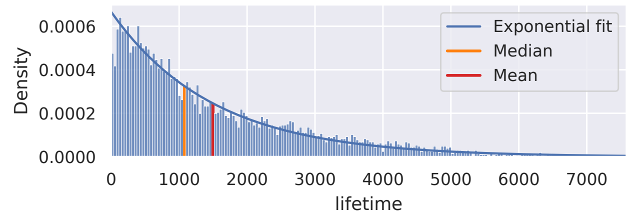


Figure 5: Distribution of vulnerability lifetimes. Equally sized bins.

distribution of the empirical values is a bit denser than the theoretical at around 5,000 days, and sparser for higher lifetimes (over 6,000 days). We assess that the exponential is still a good fit – and thus useful for calculations – for lifetimes up to around 5,000 days (98% of our data). We expected the existence of a cut-off point for the fit on the tail of the data since the exponential continues generating some small – yet non-negligible – mass for very high values (since it goes to infinity), while the lifetimes in our empirical dataset are naturally constrained by the age of the code. We then employed the Kolmogorov-Smirnov test as described in the seminal methodology of Clauset et al. [9] to statistically compare the fit of the exponential to other candidate distributions, such as the power law or the lognormal. We found the exponential to be a statistically significant better fit. Further evidence supporting the goodness of fit of the exponential can be found in Appendix A.2.

Given the above, the distribution for the lifetimes can be approximated by the probability density function below[17]:

$$f(x) = \frac{1}{1501.47} e^{-\frac{1}{1501.47}x} \quad (2)$$

[17]For most of the probability mass, except the tail (>5,000 days) as discussed above.

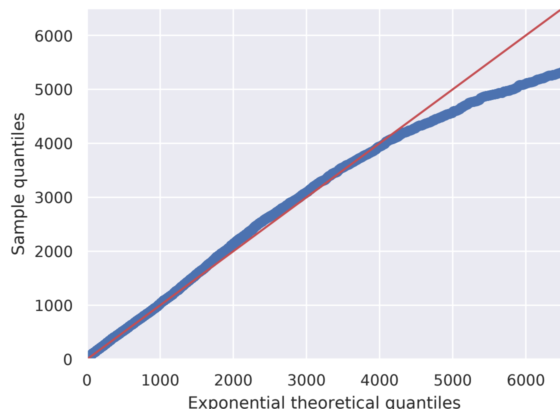


Figure 6: Q-Q Plot comparing theoretical exponential distribution and our data (blue points). The fit is excellent up to a lifetime of around 4,200 days and then gradually diverges. We can say it remains a good fit up to a lifetime of around 5,000 days.

This distribution has an average value of 1,501.47 days and a median (referred to as half-life in nuclear physics) of $ln2 \cdot 1,501.47 = 1,040.74$ days. This is the amount of time required for half of the vulnerabilities to be fixed. Conversely, 63% of vulnerabilities are fixed before the average lifetime of 1,501.47 days. Exponential distributions also provide satisfactory fits for the vulnerability lifetimes of single projects, when considered in isolation (with small variations). This can also be observed in the average and median values of Table 3. For most projects (especially the ones with many data points available), the median is close to ln2 (~0.69) times the average. According to our analysis, for all intended purposes of this paper, the empirical distribution of lifetimes in a project can be adequately approximated by an exponential distribution.

5.3 Trends over time

To investigate the progression of vulnerability lifetimes over time, we grouped CVEs by their fixing year (year of their last fixing commits, as also discussed in Section 2.2) and calculated the average lifetime for each year. Figure 7 shows how vulnerability lifetimes progressed over the years for the dataset as a whole, as well as for Firefox, Chromium and Linux. These were the projects that had enough CVEs (>20) for each year to confidently assess their lifetime over an extended period. Specifically, the grey area in the plots covers the years before the first year when at least 20 CVEs with fixing commits were available for the project.

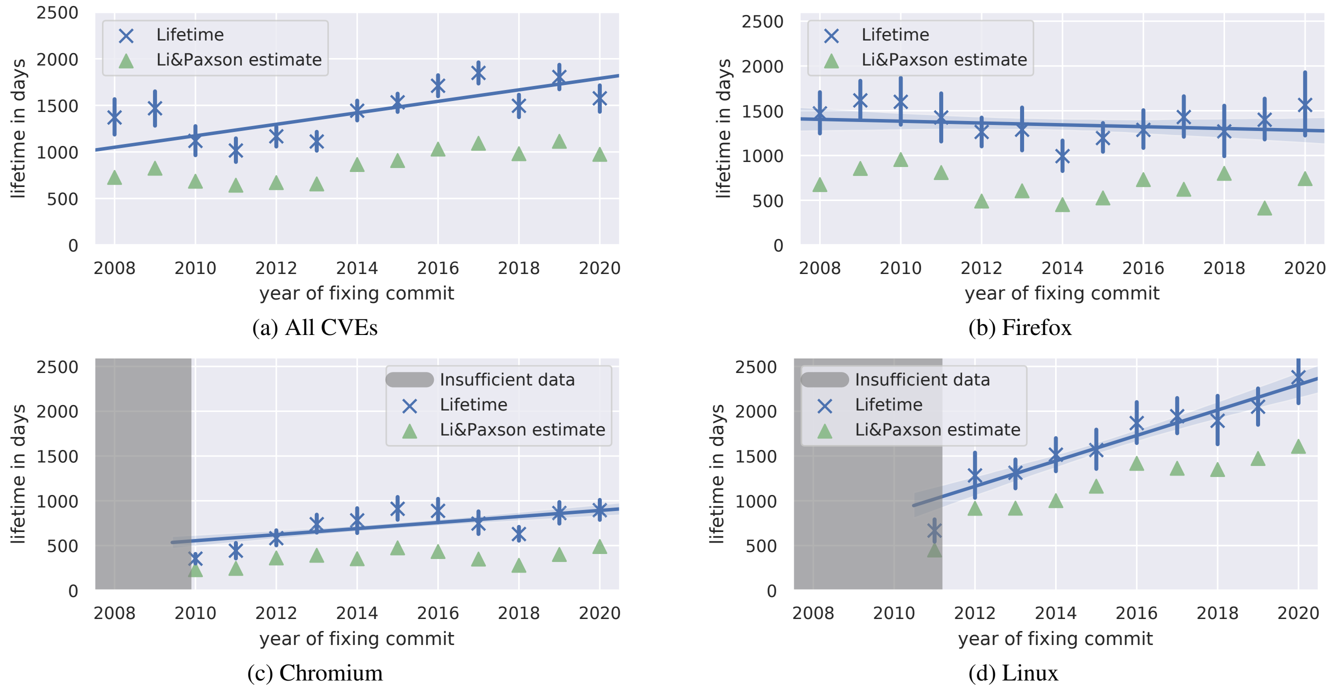
Overall (Figure 7a), vulnerability lifetimes show a sign of increase over the years with some fluctuation. When considering all CVEs, their average vulnerability lifetime increases by 42.78 days per year.

Considering the other 3 selected projects in particular, for Chromium (Figure 7c) and Linux (Figure 7d) we can observe clear increasing trends, whereas for Firefox (Figure 7b), vulnerability lifetimes are stable, even with a slight decreasing trend. It is interesting to note that although the overall increasing trends for Chromium and Linux are similar, lifetimes for Chromium can fluctuate significantly over the years, while the values for Linux fluctuate less. For the other projects that do not have enough datapoints for year-by-year analysis, we group CVEs per 2 or more years (for additional figures refer to the full version of the paper). In total, out of the 11 projects in our study: 3 (Chromium, Linux, httpd) have a clear and significant (ordinary least squares linear fit factor > 0 with 95% confidence) increasing trend; 4 (Qemu, OpenSSL, Php, Postgres) show an increase but with fewer data points available; 4 (Firefox, Wireshark, Tcpdump, FFmpeg) do not exhibit any particular trend.

Does increasing vulnerability lifetimes mean that code quality is getting worse over time? We came up with two possible conflicting explanations for increasing vulnerability lifetimes. The first is optimistic: we are fixing vulnerabilities faster than we are introducing them, and thus, there are less new vulnerabilities to find and the average age of those we are fixing is increasing, as we are "catching up". As put forward by Corbet in 2010 [11] regarding the Linux kernel "[a prominent kernel developer told me that] the bulk of the holes being disclosed were ancient vulnerabilities which were being discovered by new static analysis tools. In other words, we are fixing security problems faster than we are creating them". The pessimistic explanation is that we are introducing vulnerabilities at a similar or even greater rate than we are fixing them, and that the average age of those that are fixed is increasing, along with the age of the codebase. The optimistic explanation would require a gradual change of the shape of the distribution of lifetimes. As can be seen in Figure 8, the distribution remains exponential, with gradually increasing mean over time. Thus, the optimistic explanation is not supported by the empirical measurements, making the pessimistic explanation more likely. However, deeper investigation into the relationship between vulnerability age and code age in general is required. We present this in the next section.

5.4 Code age

To compute the overall code age of a project at a given point in time, we employed the following method. For each year X, we consider the state of the repository on the 1st of July in that year (half-way point). Subsequently, we "blame" (getting the point in time a line was last changed) every line in the repository. We consider the time-span between the last change and the half-way point to be the *regular code age* for that line in year X. To analyze the relation between regular code age and fixed vulnerable code age (vulnerability lifetime), we calculate the average code age for each year that we have vulnerability lifetime data for, and plot the result in Figure 9



(a) All CVEs (b) Firefox (c) Chromium (d) Linux

Figure 7: Average Lifetime trend (computed with our weighted average approach) for all CVEs, as well as for Firefox, Chromium and Linux, in isolation. A lower bound computed similarly to Li and Paxson's approach is included for completeness.

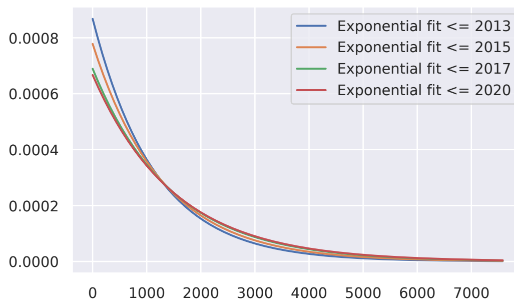


Figure 8: Distribution fit of lifetimes by year of fix

for Firefox, Chromium, Linux, and Httpd.

We observe a close correlation between average code age and average vulnerability lifetime for all projects. Both quantities have an increasing trend over time for all projects, except for Firefox, for which there is a slightly decreasing trend for both quantities. We can make two general key observations here. First, vulnerability lifetime is lower than regular code age. Second, although for most projects the spread between "vulnerable code" and "all code" appears to remain constant over time, for some projects (e.g. Chromium – see Figure 9b), this spread increases. These observations and their interpretation carry significant insights that we discuss in Section 6. But first, in the following subsection, we investigate whether there is a relation between the lifetime of a vulnerability and the type of bug that introduced it.

5.5 Types

CVE entries in the NVD are assigned a Common Weakness Enumeration identifier (CWE) [22] that denotes the type of error that led to the vulnerability. However, these identifiers, in their raw form, are not suited for studies involving multiple projects, since (a) different analysts may assign CWEs on different depths in the CWE hierarchy[18], and therefore CWEs are not directly comparable, (b) one CWE can have multiple top-level ("root") CWEs, making it difficult to compare on the root level, (c) depending on the CWE View chosen for the root level (e.g. *CWE VIEW: Research Concepts CWE-1000*), some of the CWEs in the NVD entries may not even be part of the hierarchy.

Therefore, we created a mapping between CWE identifiers and 6 custom top level categories (see Table 4) that covers the most relevant research concepts. The categories are broad enough that each CWE can be assigned to one of them, and the number of total categories is low to allow for large enough sample sizes within each.

Code Development Quality refers to vulnerabilities that are introduced due to violations of standard coding practices like infinite loops, division by zero, etc. *Security Measures* includes cryptographic issues as well as flaws related to authentication, permission and privilege management. The *Memory Management*, *Input Validation and Sanitization*, and *Concur-*

[18] CWE identifiers are organized in a hierarchical structure. More general identifiers, e.g. *CWE-682: Incorrect Calculation*, have multiple more specific "children", e.g. *CWE-369: Divide By Zero*.

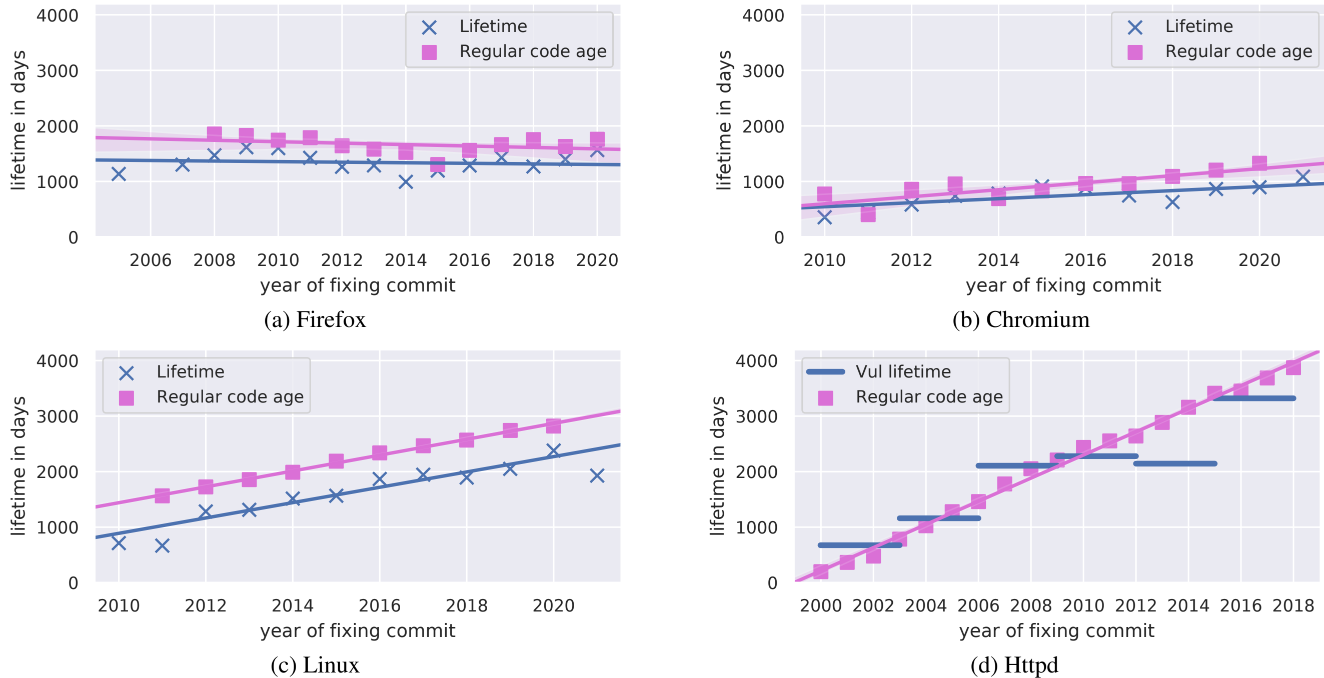


Figure 9: Age of vulnerable code vs. all code, along with linear fits, for Firefox, Chromium, Linux (kernel) and Httpd. For Httpd, vulnerability lifetimes are calculated in 4 or 5-year intervals to guarantee confidence in the estimation.

rency categories are self-explanatory. *Others* is the category that includes all CWEs that could not be matched to any of the five aforementioned categories. The exact mapping between CWE identifiers and our categories is available in Appendix A.1. The average vulnerability lifetime per category is shown in Table 4.

ID	Name	Mean	Median
5	Others	1,345.64	984.0
2	Input Validation and Sanitization	1,354.07	944.5
4	Security Measures	1,384.05	996.5
6	Concurrency	1,604.10	1,296.0
1	Memory and Resource Management	1,633.60	1,129.0
3	Code Development Quality	1,760.96	1,333.0

Table 4: Vulnerability categories and their mean and median lifetimes (in days) for all CVEs

Analysis of the data for all CVEs indicates a significant[19] difference in distribution, agreeing with previous results [18].

More thorough analysis, however, reveals that this can be attributed to differences in the prevalence of different types in different projects, rather than some deeper relation (an instance of Simpson's paradox). Specifically, for Linux, Chromium, and Firefox, no significant difference can be statistically observed[20]. As an example, the figures for Chromium are given in Table 5.

ID	Name	Mean	Median
6	Concurrency	597.43	543.0
3	Code Development Quality	647.36	700.0
2	Input Validation and Sanitization	687.92	525.0
5	Others	706.71	576.0
1	Memory and Resource Management	770.25	618.0
4	Security Measures	736.49	545.0

Table 5: Vulnerability categories of Chromium, mean and median lifetimes in days

5.6 Case study on impact of fuzzing

To show the utility of vulnerability lifetime as a metric to study issues with practical implications, we investigate the effect of automated tools (esp. fuzzing tools) on vulnerability lifetimes. Although fuzz testing in general is not a new idea, "modern" coverage-guided fuzzing with fuzzers like AFL(++)[21], libFuzzer[22] and Honggfuzz[23] (syzkaller[24] for the kernel), combined with sanitizers (e.g. (K)ASan [35], MSan[25],

[19] Kruskal-Wallis-H test with p-value of 2-91e-07 and 40% of pairwise comparisons sign. different ($\alpha = 0.05$), even with Bonferroni correction.

[20] Kruskal-Wallis-H test with p-values 0.492 (Chromium), 0.075 (Firefox) and 0.525 (Linux).

[21] https://github.com/google/AFL, https://github.com/AFLplusplus/AFLplusplus

[22] https://llvm.org/docs/LibFuzzer.html

[23] https://github.com/google/honggfuzz

[24] https://github.com/google/syzkaller

[25] https://clang.llvm.org/docs/MemorySanitizer.html

UBSan[26]) to expose latent bugs, started being widely used for FOSS in around 2016 (AFL introduced to the Linux community in 2015[27], first syzkaller talk in 2016[28], OSSFuzz launched in late 2016[29]). One may expect a measurable impact on vulnerability lifetimes by the adoption of fuzzing tools. Before approaching this question empirically, we discuss the theoretically expected impact on vulnerability lifetimes from the introduction of fuzzing tools. The introduction of fuzzers in long-lived projects would result in the discovery of some very old bugs, and thus we would expect an initial increase of the average lifetime of vulnerabilities for a short period of time. Then, considering these old bugs have been removed, we would expect continuous fuzzing to result in the discovery of bugs relatively quickly after their introduction, resulting in a drop in the average vulnerability lifetime. Overall, the expected behaviour would be a surge followed by a considerable decline.

We move on to empirically approach the question. Making the simplifying assumption that memory-related bugs are the traditional and natural targets of fuzzing, we plot the trend of Linux memory-related CVEs compared to Linux CVEs that fall into other categories (categories as presented in Section 5.5). Figure 10 shows no significant difference in the lifetime trend for the two sets of CVEs. Also, we do not observe any behaviour consistent with our expectation. It seems that the introduction of "modern" fuzzing tools did not have any noticeable impact on Linux lifetimes.

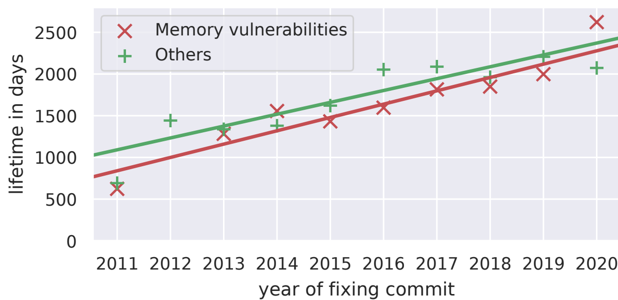


Figure 10: Lifetimes of memory-related vs. all other vulnerability categories for Linux.

Why would this be the case? We continue the investigation by looking into the five longest-surviving CVEs in our ground truth dataset for any project. Four of them are Linux CVEs (CVE-2019-15291, CVE-2019-19768, CVE-2019-11810, CVE-2019-19524), each with a lifetime of around 5,000 days (more than 13 years). All CVEs describe memory-related issues. Interestingly, 2 of them (first and last) describe issues related to USB drivers discovered by syzkaller. Fuzzing of the USB subsystem of the kernel has a rich history. Andrey Konovalov reported a batch of Linux USB vulnerabilities in 2017[30], before reporting a new batch in 2019[31] upon resumption of the USB fuzzing project. Moreover, Peng and Payer [30] (USBFuzz) used USB device emulation and coverage-guided fuzzing to discover a number of vulnerabilities in "already extensively fuzzed – versions of the Linux kernel", showing that their approach is complementary to syzkaller. The progressive discovery of vulnerabilities in this one specific subsystem of the Linux kernel showcases that the notion of "already fuzzed" may be misleading, even for relatively small subsystems of complex systems. The lack of observable impact on memory-related CVE lifetimes upon the introduction of fuzzing tools supports this assertion as very old bugs continue to get discovered with (possibly new) fuzzing tools years after fuzzing has started on the specific target. Our evidence suggests that fuzzing complex systems is not a "one-off" automated task, rather a complex ever-evolving process. A different but similar interpretation is that the "initial increase" in lifetimes that we expected due to the introduction of fuzzing is prolonged as new tools and techniques continuously enter the arsenal of testers (e.g. consider the new approaches to USB fuzzing referenced above).

None of the above imply that fuzzing in general, or as carried out in the Linux kernel, does not significantly contribute to improving security. Continues fuzzing finds large numbers of bugs (e.g. >3,000 Linux bugs discovered by syzkaller and fixed[32]). Most of these bugs are not assigned CVEs due to several reasons, e.g. they are fixed before the next release of a version or their security implications are unclear. This might be a reason why no decrease in vulnerability lifetimes is observed in our dataset, which only includes CVEs. However our techniques are of general relevance and can be used to measure the lifetimes of other bugs (e.g. discovered by syzkaller) as well. Furthermore, some of these bugs are not memory-related (e.g. KCSAN[33] is a sanitizer for Linux concurrency bugs). Overall, this case study highlights the complicated nature of fuzzing and its relation to vulnerability lifetimes. Quantifying the impact of fuzzing in improving the security of codebases and its impact on lifetimes is still an open problem that needs further investigation.

6 Implications and discussion

Is software getting more secure over time? In Section 5, we presented a number of results related to this question.

[26] https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html
[27] https://lwn.net/Articles/657959/
[28] https://github.com/google/syzkaller/blob/master/docs/talks.md
[29] https://opensource.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html
[30] https://www.openwall.com/lists/oss-security/2017/12/12/7
[31] https://www.openwall.com/lists/oss-security/2019/08/20/2
[32] https://syzkaller.appspot.com/upstream/fixed
[33] https://github.com/google/ktsan/wiki/KCSAN

We established that there is no evidence to support that we are introducing (and consequently fixing) significantly fewer new vulnerabilities over time. We also made the following observations: (a) vulnerability lifetimes are on average lower than regular code age, and (b) for some projects this spread increases over time.

Observation (a) shows that vulnerability-fixing has a sizable benefit in reducing the window of exposure to zero-day attacks. If this were not the case (if vulnerability lifetimes were similar or greater than regular code age), one could say that our vulnerability-finding practices would be totally ineffective – fortunately this is not true.

Observation (b), that for some projects vulnerability lifetimes, on average, increase slower over time compared to code age, is more interesting. One explanation could be that we are (introducing and) fixing an increasingly large number of new vulnerabilities each year, in addition to a few very old ones. Figure 8 does not support this explanation, since overall the distribution of vulnerability lifetimes does not significantly change and remains exponential, slowly stretching towards higher means over time. The alternative interpretation would be that there are parts of code that mature, i.e. we do not find vulnerabilities affecting them anymore, and apart from these parts we continue finding a similar ratio of new compared to old vulnerabilities. Our data support this interpretation. Overall, this interpretation suggests that we could be slowly progressing towards a state of relative maturity, where vulnerability lifetimes become stable over time and not correlated to code age, even if the latter is increasing. In this state, the older parts of the codebase will be hardened, and we will be finding vulnerabilities introduced in a possibly large, yet bounded, time period. One could challenge this interpretation with the argument that vulnerabilities in older parts of the code are not found because nobody is looking for them. Although new parts of code may come under additional manual scrutiny, we do not believe that the vulnerability hunting process is differs drastically between old and even-older parts of the code. It would be beneficial to repeat the measurement some years from now, in order to see whether the predicted behavior of a stop to the constant increase of vulnerability lifetimes will hold for the projects under question.

> Overall, even though we may not be decreasing the number of vulnerabilities in a given codebase, there are indications that we could be making progress towards achieving a notion of maturity, where vulnerabilities will be mostly absent from code older than a specific point in the past.

Are vulnerabilities (in a given project) equal? When testing for differences in vulnerability lifetimes for all CVEs (all projects), we found that there exist statistically significant differences, even when considering our custom categories. This result is in line with the observations of Li and Paxson [18]. They stopped their investigation at this point, however our dataset allowed us to explore further and investigate whether the relative frequency of different vulnerability types in different projects is the cause of the observation above. Indeed, we found no statistically significant evidence supporting a relationship between the lifetime of a vulnerability and its type, within a project. For example, Category 3 has the highest average lifetime in Table 4 (1,760 days for all CVEs) but by far the lowest in Table 8 (752 days for Firefox). We attribute differences observed in previous studies to differences in the ratios of different types in different projects. Therefore, the notion that some vulnerability categories are in general harder to find than others (e.g. memory bugs are harder to find than input validation bugs), is not supported by our findings.

> Different vulnerability categories seem to be equally difficult to find (at least post release); overall, our results are consistent with the view that all vulnerabilities in a project are equal, and their order of discovery is random.

Can we compare? Meaningful quantitative security metrics are notoriously difficult to arrive at [38]. Metrics that can meaningfully be used to compare different products/projects are especially rare. Simply comparing vulnerability lifetimes or trends in lifetimes between projects is not suitable, as they can be heavily correlated to regular code age.

> We put forward the hypothesis that the following two metrics may be helpful for comparative studies: (a) the spread between overall code age and vulnerability lifetime, or alternatively the ratio between average vulnerability lifetime and code age; (b) the rate of change (increase or decrease) of the spread between overall code age and vulnerability lifetime.

Further investigation of these metrics as comparison instruments is a very interesting avenue for future research.

How much fuzzing is enough? Traditionally, security researchers tend to focus on new, relatively less tested parts of the code to test for vulnerabilities. Recently, Zhu and Böhme [42] came to the conclusion that with limited resources, fuzzing code that has recently changed is the best vulnerability discovery strategy. Our results support this statement to an extent, in the sense that the distribution of vulnerability lifetimes can be described by a decreasing function (exponential – see Figure 5). However, a significant number of vulnerabilities have large lifetimes and fuzzers keep discovering very old vulnerabilities for years (see Section 5.6). Taking into account the well-known asymmetries of computer security, finding these vulnerabilities that potentially impact many legacy systems, is also important. Furthermore, further

focused research is required in order to better understand the impact of fuzzing in improving the security of codebases and its impact on vulnerability lifetimes.

> Overall, fuzzing old code seems to still produce results even for "extensively tested" targets. Further research is needed to understand and quantify the impact of fuzzing and its relation to lifetimes.

7 Threats to validity

Dataset. The data in vulnerability databases often experience bias from several sources [7].
– Completeness. Although the NVD is one of the largest collections of software vulnerabilities, it can not be considered complete, since many vulnerabilities may never get a CVE. Further research is required to investigate the lifetime of those vulnerabilities. Furthermore, although we strove to map as many CVEs to their fixing commits as possible, our approach is not able to identify a fixing commit for every single CVE affecting a program. However, the dataset we gathered is big and complete enough to be representative, and we do not expect our results on vulnerability lifetimes and their characteristics to be significantly influenced by some missing data points.
– Correctness. Entries in the NVD are manually curated and analyzed, but might still include errors. Additionally, mistakes in the commit message when including the bug ID or a CVE-ID can lead to incorrect mappings. We corrected these errors during our data cleaning process, however, some errors may have evaded detection. We expect these to be few and to not affect our general observations.
– Generalization. Although we do not claim validity of our results for other projects, apart from the 11 we included in our dataset, we believe that the selected projects are a large representative sample and the insights gained from our results are, to an extent, of general significance.
Independence. Some of our arguments in the Results section rely on the implied assumption that vulnerability lifetimes are independent. It has been shown that some vulnerability discovery events can be dependent [29], either due to a new class of vulnerabilities being discovered, a new tool being made available, or a new area of code coming under scrutiny. These dependencies manifest themselves as small bursts in the vulnerability discovery rate and are a particular problem to vulnerability discovery models that try to model the time between discoveries. It is difficult to imagine (let alone test) how such dependencies would affect the lifetimes of vulnerabilities, especially in a large and diverse dataset like ours. Also, our empirical results do not indicate the existence of any kind of dependency regarding vulnerability lifetimes. Therefore, we consider the independence of vulnerability lifetimes to be a reasonable assumption. Some points in the Discussion section also imply an assumption of independence of discovery events. Again, small bursts of discoveries may exist due to dependent discoveries, however they do not affect the arguments being made, which describe large-scale behaviors.
Heuristic error. Our main findings are consistent over different projects and over time. Therefore, we do not believe the error of our lifetime-estimation heuristic (as discussed in Section 4.2) to affect their validity.

8 Conclusion

In this paper we studied the lifetimes of vulnerabilities. A vulnerability's lifetime is the amount of time it remains in the codebase of a software project before it is discovered and fixed. Via a rigorous process we showed that it is possible to accurately compute the metric (lifetime) automatically when enough data points are available, via a heuristic code analysis technique. Our technique is of general relevance and can be used to study lifetimes of bugs and vulnerabilities for a wide variety of software. We also showed that measurements using the metric can have theoretical and practical implications. Thus, we believe vulnerability lifetime to be a promising software security metric.

Further research is required to better understand how the metric can be used to quantify the impact of automated tools on the security of codebases, as well as how vulnerabilities not assigned CVEs affect the results of the measurement. Moreover, further investigation of the theoretical implications of the metric w.r.t. vulnerability discovery models and software reliability models in general, could provide interesting insights.

Acknowledgments

We would like to thank our shepherd Zhiyun Qian and the anonymous reviewers for helping us significantly improve the paper. This work has been co-funded by the German Federal Ministry of Education and Research and the Hessen State Ministry for Higher Education, Research and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE.

References

[1] Nikolaos Alexopoulos, Sheikh Mahbub Habib, Steffen Schulz, and Max Mühlhäuser. The tip of the iceberg: On the merits of finding security bugs. *ACM Trans. Priv. Secur.*, 24(1), September 2020.

[2] Omar H. Alhazmi and Yashwant K. Malaiya. Modeling the vulnerability discovery process. In *16th International Symposium on Software Reliability Engineering (ISSRE 2005), 8-11 November 2005, Chicago, IL, USA*, pages 129–138. IEEE Computer Society, 2005.

[3] Omar H Alhazmi and Yashwant K Malaiya. Quantitative vulnerability assessment of systems software. In *Annual Reliability and Maintainability Symposium, 2005. Proceedings.*, pages 615–620. IEEE, 2005.

[4] Omar H. Alhazmi and Yashwant K. Malaiya. Measuring and enhancing prediction capabilities of vulnerability discovery models for apache and IIS HTTP servers. In *17th International Symposium on Software Reliability Engineering (ISSRE 2006), 7-10 November 2006, Raleigh, North Carolina, USA*, pages 343–352. IEEE Computer Society, 2006.

[5] Leyla Bilge and Tudor Dumitras. Before we knew it: an empirical study of zero-day attacks in the real world. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 833–844. ACM, 2012.

[6] Canonical. Ubuntu cve tracker. https://git.launchpad.net/ubuntu-cve-tracker. Accessed: 2020-03-18.

[7] Steve Christey and Brian Martin. Buying into the bias: Why vulnerability statistics suck, 2013. Presentation at BlackHat, Las Vegas, USA, slides available at https://media.blackhat.com/us-13/US-13-Martin-Buying-Into-The-Bias-Why-\Vulnerability-Statistics-Suck-Slides.pdf.

[8] Sandy Clark, Stefan Frei, Matt Blaze, and Jonathan Smith. Familiarity breeds contempt: The honeymoon effect and the role of legacy code in zero-day vulnerabilities. In *Proceedings of the 26th annual computer security applications conference*, pages 251–260. ACM, 2010.

[9] Aaron Clauset, Cosma Rohilla Shalizi, and Mark EJ Newman. Power-law distributions in empirical data. *SIAM review*, 51(4):661–703, 2009.

[10] Kees Cook. Security bug lifetime. https://outflux.net/blog/archives/2016/10/18/security-bug-lifetime/, 2016.

[11] Jonathan Corbet. Kernel vulnerabilities: old or new? https://lwn.net/Articles/410606/, 2010.

[12] Debian. Debian security tracker. https://salsa.debian.org/security-tracker-team/security-tracker/-/tree/master/data/CVE.

[13] Alexandre Dulaunoy. Cve-search. https://github.com/cve-search/cve-search, Jan 2020.

[14] Stefan Frei. *Security econometrics: The dynamics of (in) security*. PhD thesis, ETH Zurich, 2009.

[15] HyunChul Joh, Jinyoo Kim, and Yashwant K. Malaiya. Vulnerability discovery modeling using weibull distribution. In *19th International Symposium on Software Reliability Engineering (ISSRE 2008), 11-14 November 2008, Seattle/Redmond, WA, USA*, pages 299–300. IEEE Computer Society, 2008.

[16] Jinyoo Kim, Yashwant K. Malaiya, and Indrakshi Ray. Vulnerability discovery in multi-version software systems. In *Tenth IEEE International Symposium on High Assurance Systems Engineering (HASE 2007), November 14-16, 2007, Dallas, Texas, USA*, pages 141–148. IEEE Computer Society, 2007.

[17] Brian Krebs. Why counting flaws is flawed. https://krebsonsecurity.com/2010/11/why-counting-flaws-is-flawed/, 2010.

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

[19] Nicholas Luedtke. linux kernel cves. https://github.com/nluedtke/linux_kernel_cves.

[20] Andrew Meneely, Harshavardhan Srinivasan, Ayemi Musa, Alberto Rodriguez Tejeda, Matthew Mokary, and Brian Spates. When a patch goes bad: Exploring the properties of vulnerability-contributing commits. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, Baltimore, Maryland, USA, October 10-11, 2013*, pages 65–74, 2013.

[21] Andy Meneely. Vulnerability history project. https://github.com/VulnerabilityHistoryProject. Accessed: 2020-03-18.

[22] MITRE. Common weakness enumeration. https://cwe.mitre.org/data/definitions/699.html, 2020.

[23] 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 *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 692–708, 2015.

[24] Viet Hung Nguyen and Fabio Massacci. The (un)reliability of NVD vulnerable versions data: an empirical experiment on google chrome vulnerabilities. In Kefei Chen, Qi Xie, Weidong Qiu, Ninghui Li, and

Wen-Guey Tzeng, editors, *8th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '13, Hangzhou, China - May 08 - 10, 2013*, pages 493–498. ACM, 2013.

[25] U.S. National Institute of Standards and Technology. National vulnerability database. https://nvd.nist.gov/home.

[26] Andy Ozment. Software security growth modeling: Examining vulnerabilities with reliability growth models. In Dieter Gollmann, Fabio Massacci, and Artsiom Yautsiukhin, editors, *Quality of Protection - Security Measurements and Metrics*, volume 23 of *Advances in Information Security*, pages 25–36. Springer, 2006.

[27] Andy Ozment. Improving vulnerability discovery models. In Günter Karjoth and Ketil Stølen, editors, *Proceedings of the 3th ACM Workshop on Quality of Protection, QoP 2007, Alexandria, VA, USA, October 29, 2007*, pages 6–11. ACM, 2007.

[28] Andy Ozment and Stuart E. Schechter. Milk or wine: Does software security improve with age? In Angelos D. Keromytis, editor, *Proceedings of the 15th USENIX Security Symposium, Vancouver, BC, Canada, July 31 - August 4, 2006*. USENIX Association, 2006.

[29] James Andrew Ozment. *Vulnerability discovery & software security*. PhD thesis, University of Cambridge, 2007.

[30] Hui Peng and Mathias Payer. Usbfuzz: A framework for fuzzing USB drivers by device emulation. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 2559–2575. USENIX Association, 2020.

[31] 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 *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 426–437, 2015.

[32] Valentina Piantadosi, Simone Scalabrino, and Rocco Oliveto. Fixing of security vulnerabilities in open source projects: A case study of apache HTTP server and apache tomcat. In *12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019, Xi'an, China, April 22-27, 2019*, pages 68–78. IEEE, 2019.

[33] Eric Rescorla. Is finding security holes a good idea? *IEEE Secur. Priv.*, 3(1):14–19, 2005.

[34] B Schneier. Cryptogram september 2000-full disclosure and the window of exposure. https://www.schneier.com/crypto-gram/archives/2000/0915.html, 2000.

[35] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In Gernot Heiser and Wilson C. Hsieh, editors, *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, pages 309–318. USENIX Association, 2012.

[36] Muhammad Shahzad, Muhammad Zubair Shafiq, and Alex X. Liu. A large scale exploratory analysis of software vulnerability life cycles. In Martin Glinz, Gail C. Murphy, and Mauro Pezzè, editors, *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 771–781. IEEE Computer Society, 2012.

[37] Lin Tan, Chen Liu, Zhenmin Li, Xuanhui Wang, Yuanyuan Zhou, and ChengXiang Zhai. Bug characteristics in open source software. *Empirical Software Engineering*, 19(6):1665–1705, 2014.

[38] Vilhelm Verendel. Quantified security is a weak hypothesis: a critical survey of results and assumptions. In Anil Somayaji and Richard Ford, editors, *Proceedings of the 2009 Workshop on New Security Paradigms, Oxford, United Kingdom, September 8-11, 2009*, pages 37–50. ACM, 2009.

[39] Martin B Wilk and Ram Gnanadesikan. Probability plotting methods for the analysis for the analysis of data. *Biometrika*, 55(1):1–17, 1968.

[40] Sung-Whan Woo, Omar H. Alhazmi, and Yashwant K. Malaiya. Assessing vulnerabilities in apache and IIS HTTP servers. In *Second International Symposium on Dependable Autonomic and Secure Computing (DASC 2006), 29 September - 1 October 2006, Indianapolis, Indiana, USA*, pages 103–110. IEEE Computer Society, 2006.

[41] Limin Yang, Xiangxue Li, and Yu Yu. Vuldigger: A just-in-time and cost-aware tool for digging vulnerability-contributing changes. In *GLOBECOM 2017-2017 IEEE Global Communications Conference*, pages 1–7. IEEE, 2017.

[42] Xiaogang Zhu and Marcel Böhme. Regression greybox fuzzing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, CCS 2021 (to appear)*, 2021.

A Appendix

A.1 Vulnerability categories

Mappings to top-level categories
Below are the mappings from CWEs to our own top level categories:
1. Memory and Resource Management
2. Input Validation and Sanitization
3. Code Development Quality
4. Security Measures
5. Others
6. Concurrency

mappings = {CWE-20: 2, CWE-189: 4, CWE-119: 1, CWE-125: 1, CWE-399: 1, CWE-NVD-Other: 5, CWE-200: 4, CWE-476: 1, CWE-264: 4, CWE-416: 1, CWE-835: 3, CWE-NVD-noinfo: NaN, CWE-362: 6, CWE-400: 1, CWE-787: 1, CWE-772: 1, CWE-310: 4, CWE-190: 1, CWE-74: 2, CWE-17: 3, CWE-284: 4, CWE-415: 1, CWE-369: 3, CWE-19: 5, CWE-834: 3, CWE-79: 4, CWE-754: 5, CWE-674: 3, CWE-120: 1, CWE-94: 2, CWE-388: 5, CWE-269: 4, CWE-254: 4, CWE-129: 2, CWE-287: 4, CWE-617: 3, CWE-276: 4, CWE-404: 1, CWE-134: 5, CWE-862: 4, CWE-320: 4, CWE-89: 2, CWE-347: 4, CWE-682: 3, CWE-16: 5, CWE-665: 5, CWE-755: 5, CWE-732: 4, CWE-311: 4, CWE-770: 1, CWE-252: 5, CWE-534: 5, CWE-704: 5, CWE-22: 2, CWE-532: 5, CWE-193: 3, CWE-843: 5, CWE-391: 5, CWE-191: 1, CWE-59: 2, CWE-763: 1, CWE-358: 4, CWE-285: 4, CWE-863: 4, CWE-77: 2, CWE-327: 4, CWE-330: 5, CWE-295: 5, CWE-352: 5, CWE-92: 4, CWE-664: 1, CWE-93: 2, CWE-275: 4, CWE-434: 5, CWE-707: 2, CWE-668: 4, CWE-361: 6, CWE-319: 4, CWE-255: 4, CWE-824: 1, CWE-1187: 1, CWE-426: 4, CWE-417: 5, CWE-427: 5, CWE-610: 5, CWE-522: 4, CWE-345: 5, CWE-354: 5, CWE-91: 2, CWE-918: 5, CWE-922: 4, CWE-706: 5, CWE-538: 4, CWE-290: 4, CWE-601: 4, CWE-346: 5, CWE-502: 2, CWE-1021: 5, CWE-78: 2, CWE-199: 5, CWE-829: 5, CWE-281: 4, CWE-203: 4, CWE-401: 1, CWE-908: 1, CWE-667: 1, CWE-209: 4, CWE-88: 2, CWE-459: 1, CWE-326: 4, CWE-270: 4, CWE-331: 5, CWE-122: 1, CWE-367: 6, CWE-909: 1, CWE-552: 4, CWE-436: 5, CWE-131: 1, CWE-672: 1, CWE-271: 4, CWE-681: 3, CWE-212: 4}

A.2 Lifetime Distribution

Kolmogorov-Smirnov tests. We list the results of the comparison using the seminal methodology of Clauset et al. [9] in Table 6. The exponential distribution is a significantly better fit than other candidate distributions.

Distribution	R
powerlaw	9203.49
lognormal	506.33
truncated power law	5505.75
lognormal positive	506.33

Table 6: Comparison of distribution fits with exponential. Positive R-values mean that the exponential is a better fit. All comparisons are at a significance level of at least 99%.

Comparative probability table Table 7 is a comparative probability table to numerically convey the goodness-of-fit of the exponential distribution and its usefulness in calculations.

Lifetime	Theoretical CDF	Empirical CDF
188	0.1171	0.1
376	0.2210	0.2
562	0.3117	0.3
791	0.4090	0.4
1,081	0.5128	0.5
1,436	0.6154	0.6
1,927	0.7226	0.7
2,574	0.8197	0.8
3,520	0.9040	0.9

Table 7: Numerical comparison of empirical and theoretical CDF. Values are chosen as the quantiles of the empirical data. The empirical distribution does not deviate more than 2.5 percentage points from the theoretical one.

A.3 Vulnerability types per project

For completeness, we present the average and median lifetimes per vulnerability category for Firefox (Table 8) and the Linux kernel (9).

ID	Name	Mean	Median
3	Code Development Quality	714.93	459.0
5	Others	1,116.22	977.0
6	Concurrency	1,170.27	1,137.0
4	Security Measures	1,284.52	1,139.0
1	Memory and Resource Management	1,303.23	954.5
2	Input Validation and Sanitization	1,409.22	1,149.5

Table 8: Vulnerability categories for Firefox, mean and median lifetime in days

ID	Name	Mean	Median
2	Input Validation and Sanitization	1,534.54	1,291.0
4	Security Measures	1,660.90	1,292.0
5	Others	1,681.83	1,166.0
1	Memory and Resource Management	1,756.77	1,390.5
3	Code Development Quality	1,858.60	1,517.5
6	Concurrency	1,904.62	1,663.5

Table 9: Vulnerability categories for Linux, mean and median lifetime in days