tbn

Bachelor thesis by Manuel Brack

1. Review: Prof. Dr. Max Mühlhäuser
2. Review: Nikolaos Alexopoulos
Darmstadt

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Computer Science
Department
Telecooperation Lab
SPIN

Contents

Abstract

1 Introduction

- *Security bugs are very dangerous*
- *A lot of effort is expended on finding and fixing them (dynamic and static analysis, bug bounties, etc.). Also developers and big development teams are trained in security nowadays and code undergoes several stages of review (see e.g. at Google or at other big companies).*
- *One interesting metric that has been used in previous research ... to... is the lifetime of a vulnerability, meaning...*
- *In this thesis, I investigate this metric along two major axes (a) time, and (b) different projects (development teams).*
- *Results may shed light on important aspects of the development and vulnerability discovery processes, such as...*

1.1 Background

Investigating characteristics of software vulnerabilities has proven to be a valuable resource for addressing questions of software security and the development process in general. Additional insight can be provided by combining this information with the analysis of the underlying code base. By analyzing the characteristics a vulnerability-introducing change might have, for example, tools for vulnerability assessment and prediction might be enhanced. This also facilitates the investigation of the patch development cycle and the analysis of variations between different classes of issues with respect to their characteristics in the code base.
Such work does not only provide specific insights for researchers, but can also result in recommendations for the secure development of software products in general.

In this thesis we focus on evaluating software vulnerabilities and the corresponding code with respect to the lifetime of vulnerabilities. We believe that this metric can provide valuable insights on the evolution of software security and code quality in the last years. More conventional metrics like the number of vulnerabilities found in a specific year may be affected by numerous external factors that bias the results. The lifetime of vulnerabilities fixed in that year, on the other hand, does not directly correspond to the resources committed to finding vulnerabilities. Therefore, it can be considered a more genuine indicator of code quality.
Furthermore, this metric allows for a more unbiased comparison between different software products. Again, metrics like the number of vulnerabilities discovered in a certain time period, for example, are often not representative. It does not account for the amount of changes that have been made. Naturally, a

software project that underwent more changes in that time period is more likely to have introduced vulnerabilities as unwanted result of these modifications. An often considered adaptation is to compare the ratio of vulnerabilities introduced per lines of code changed, but this may also produce skewed results. For example, implementing a new visual feature in a browser may very quickly introduce thousands of lines of code while not being relevant for security related changes. Moreover, some software products receive significantly more attention from people looking for vulnerabilities than others. This, combined with very different numbers of regular users of the product, leads to a reporting bias often resulting in higher percentages of vulnerabilities being discovered for actively researched and investigated projects. We think that the lifetime of vulnerabilities, however, depends mostly on the development and code review process of the software developer itself . Thus it may offer a less biased view on software quality and security than other metrics associated with vulnerability investigation. In this thesis we will asses if our metric is suitable for comparing the code quality and security of different software products across varying fields of application, sizes of development teams and types of software.

2 Related Work

3 Problem Definition & Research Questions

Todo: Introduction sentence

3.1 Problem Definition

Defining the lifetime of a software vulnerability is a non-trivial problem. First of all we consider a generic model for the life-cycle of software vulnerabilities. Browne *et al.* ([2]) identifies the following steps:

1. A flaw is introduced to the software and then deployed for widespread use
2. The flaw is discovered by either the vendor itself or a third party
3. An exploit for the flaw is developed, now making it a vulnerability
4. The vulnerability is made public and therefore exploitable by a large number of parties
5. A patch or revision for the vulnerability is released, preventing any exploits
6. The vulnerability dies when there are no more instances of the software affected by the underlying flaw

Of course, these events might occur in different order. Following typical public disclosure procedures the developer of a software is able to identify and fix a flaw before the existence of the vulnerability is publicly disclosed. This ensures that a fixing patch can be released before a wide-scale exploit of the vulnerability is possible.
However, it is clear that this definition of a vulnerability's life cycle allows for multiple definitions of *vulnerability lifetime*. The time a vulnerability affects the stable version can be one possible definition. That is, the time span between the deployment of the first patch containing the exploitable flaw and the release of the patch fixing that vulnerability. Alternatively, a vulnerability could be considered 'alive' as long as there are still devices running a version of the software susceptible to the corresponding exploit. In this case the lifetime of the vulnerability would be the time span between the first deployment of the patch introducing the flaw and the point in time no device is affected by a vulnerability anymore. Either because all devices upgraded to a version of the software containing the fixing patch or because the vulnerable software was deleted. While this aspect is interesting the necessary information for this definition is, of course, almost impossible to obtain.
In this thesis, however, we focus on investigating the software development processes concerning vulnerabilities, rather than the patch distribution or public exposure. In consequence, we evaluate the time the underlying flaw of the vulnerability is present in the code base. Arguably, this time does not necessarily reflect on the assessment of a software product's security. Different projects, for example, might also have different release cycles. That means that a vulnerability remaining in the code for 3 months might be guaranteed to be deployed in a stable version of one project while another project might only release a stable version

twice a year, thus fixing the vulnerability before it was ever deployed. Furthermore, two vulnerabilities with the same lifetime might have been present in a deployed version for different amounts of time. Therefore, this metric might be consider more valuable for indicating code quality rather than safety. However, previous research suggests that vulnerabilities may remain in the code for months and years. Thus making the portion of their lifetime they spend in development only version of the software somewhat negligible.

In order to perform our evaluation we link a vulnerability with one or more **fixing-commits** in the software's code repository. As the name indicates this/these commit(s) resolve(s) the flaw in the code that caused the vulnerability.

Definition 1. *A fixing-commit of a software vulnerabilities (partially) solves the underlying software flaw that causes the vulnerability.*

From this fixing-commit we then identify one or more **vulnerability contributing commits (VCCs)**. These are the changes in the code that introduced the faulty code.

Definition 2. *A VCC introduced changes into the code of a software project that led to an exploitable vulnerability.*

Defining which changes contributed to a vulnerability and can therefore be considered a VCC is a non-trivial task. Pin-pointing the introduction of vulnerability to one specific commit is even more challenging. Modern software projects contain tens and hundreds of thousands of commits, thus making this a difficult and potentially very time-consuming task. In some cases only the combination of two independent changes may lead to the creation of vulnerability. However, this work does not rely on identifying a specific commit as VCC, as we are not interested in specific aspects of that commit. For us it is sufficient to estimate the point in time the vulnerability was introduced. With respect to this model we define the lifetime of a software vulnerability as follows:

Definition 3. *The lifetime l of a software vulnerability in the code base is described by the time span between the latest fixing-commit and the earliest VCC.*

We chose to assume the *worst-case scenario* for our evaluation. To that end this definition of a vulnerability's lifetime is prone to overestimate rather than underestimate the correct value. This becomes clear when considering vulnerabilities with multiple fixing-commits and/or VCCs. In cases of multiple VCCs we will choose the date of the earliest VCC which would imply that the code was already vulnerable after the introduction of that change. Furthermore, we we will also choose the latest multiple fixing commits, indicating that the code was still vulnerable after the introduction of earlier fixing commits.

3.2 Research Questions

With respect to this metric and the extensive amount of data collected, we hope to answer multiple questions regarding the quality and security of software projects.

Earlier research suggests that software vulnerabilities remain in the code base for months and years on average. Li *et al.* ([11]) computed a median lifetime 438 days, evaluating over 3000 vulnerabilities in 682 open source projects. They, however, used a heuristic that underestimates the actual lifetime and, consequently,

only provides a lower bound. To that end, we expect our values to be significantly higher. In 2001 and 2006 *Chou et al.* ([3]) and *Ozment et al.* ([14]) evaluated the average lifetimes for the Linux and OpenBSD Kernel, respectively. *Chou et al.* came up with 1.8 years for Linux and Ozment *et al.* estimated at least 2.6 years for OpenBSD. Jonathan Corbet [6] and Kees Cook[4] did a similar analysis for the Linux Kernel in 2010 and 2016, respectively. Both concluded that vulnerabilities remain in the Linux source code for about 5 years. Since then software projects have become increasingly complex and it is possible that the implemented quality and security control mechanism have not been able to keep up the pace. We, therefore, expect the average lifetimes to be at least around 2 to 2.5 years, but presume that they could be notably higher. We formulate this as the following research question:

RQ 1. *How long do software vulnerabilities live in the code base?*

Any major differences in the average of lifetimes, compared to previous research, would indicate that there has been a significant change over the last years. As mentioned before we assume that the average lifetime might have increased. This leads to the subsequent research question:

RQ 2. *Does the lifetime of software vulnerabilities change over time?*

In an ideal world more severe vulnerabilities would be fixed faster than less serious ones. This, however, does not necessarily reflect on the actual lifetime of more severe vulnerabilities in the code base. Li and Paxson ([11]) found no monotonic correlation between the lifetime and severity of a vulnerability. We expect to be able to reproduce these results on our larger data set and with increased accuracy of the lifetime approximation and introduce the resulting question:

RQ 3. *Do more severe vulnerabilities remain in the code base for shorter amounts of time?*

When considering different types of vulnerabilities, intuition suggests that some types might be easier to locate and resolve than others. Missing input sanitation for example seems to be easier to fix than a complex race condition for example. Additionally, vulnerabilities related to issues on online services may be discovered quicker, as at least one of thousands of daily users is likely to encounter that flaw. These assumptions are also supported by the results of Li and Paxson ([11]) who observed drastic differences in lifetimes between different types.

RQ 4. *Do different types of vulnerabilities have different lifetimes?*

The distribution of lifetimes may also provide valuable insights on the characteristics of vulnerability life cycles and the effectiveness currently implemented measures. The results of Li and Paxson indicate that the lifetimes might be somewhat log-normal or power-law distributed.

RQ 5. *What distribution do vulnerability lifetimes follow?*

As mentioned before, this approach also considers vulnerabilities that did not affect the stable version. But for some vendors, Debian for example, it is part of their security strategy to only include changes in the stable version after a significant time of testing. This *freeze* should have an impact on the factors mentioned before. This leads to the following two research questions:

RQ 6.1. *Does the stable freeze have a positive impact on the considered aspects?*

RQ 6.2. *How long should the stable freeze be ideally?*

4 Methodology

TODO: Introduction sentence

4.1 Data Sources

In order to analyze the lifetime of software vulnerabilities with respect to different vulnerability and software aspects, we require two sets of data. Our data collection process is based on the information provided by the National Vulnerability Database (NVD) [13] of the U.S. National Institute of Standards and Technology (NIST). The NVD is a manually curated database that lists publicly disclosed vulnerabilities. It also contains corresponding information about affected software and respective versions, type and severity of the vulnerability as well as references to third party information. Each vulnerability is assigned a unique identifier called Common Vulnerabilities and Exposure (CVE) ID. We use the CVE-Search tool [8] that creates a local copy of the NVD which allows for easy and automated querying and collection of CVE data.
For each software product included in this thesis we also gather data from the respective source code repository. While not every development team uses git for the official development process we were able to find an up-to-date git clone for all of them. This allows for utilizing the same features of git for all repositories during out data collection.
Additionally, we included data from other sources like researchers, software security experts and security advisories to obtain as much information as possible. Each of them is explicitly introduced in section 4.4.

4.2 Vulnerability Mapping

As discussed in 3.1 it is necessary to link CVEs to fixing commits. In order to get the largest possible data-set we apply and combine multiple techniques and approaches.

1. **Commit mentioning CVE**
   Some commit message already contain the identifier of a CVE and can easily be linked to the corresponding vulnerability.
2. **CVEs mentioning commit**
   In some cases after a fixing-commit resolving a specific CVE is published a link to that commit might be added to the CVE's reference list. From that hyperlink a unique commit identifier (e.g. a hash) can be determined and then linked to the corresponding commit in the repository.

3. **CVE and commit mentioning common bug identifier**
   Many of the considered repositories have a mature bug tracking system and a stringent policy on mentioning a bug identifier in each fixing commit. Furthermore, there usually is a specification regarding the syntax of said bug ID notations. Once this syntax is known, fixing commits can be linked to identifiers of the vendors bug tracking system using simple regular expression matching.

   CVEs on the other hand may contain references to the public website of these bug tracking systems. These links can be identified using regular expression matching and the bug ID can be extracted afterwards.

4. **Third party mappings**
   Fellow researchers and other security experts have already gathered and structured information linking CVEs and fixing commits. In addition to our own analysis we incorporate this data, especially in cases where the desired information can not be acquired automatically.

Combining these methods allowed us to create an extensive database linking over 4400 CVEs to fixing commits across 10 different open source projects. To the best of our knowledge this is one of the largest data collections of its kind and we will make it publicly available for other researchers to use.
While the number of vulnerabilities is high the number of included projects might appear low. However, there are simply not that many software projects with a significantly large number of entries in the NVD. This becomes clear when looking at the top 10 software products by numbers of associated CVEs in Li and Paxons's work [11]. The project with the 10th most entries in their subset of all projects only has 77 associated CVEs. Rather than including more software projects into our evaluation we focused on achieving a high percentage of CVEs mapped to a fixing commit for projects with a large number of entries in the NVD. We, therefore, determined that this approach was better suited to achieve an overall larger data set. Additionally, this thesis not only evaluates the imposed research questions over the complete data set, but also compares the results of different software projects with each other. This, on the other hand, is only possible for software projects that have a large enough number of data points to be considered statistically robust. This renders many projects unsuitable for this work as discussed later in section 4.4. This also makes our data-set the possibly most complete one for the projects considered.
While this approach provides valuable and extensive information on software vulnerabilities and the corresponding changes in the code, there are some limitation that have to be considered:

- **Completeness:** Although the NVD does provide vulnerability information on a large-scale, it can not be considered complete. Additionally, not all CVEs can be mapped to a fixing-commit with the techniques introduced above. A vulnerability in a third-party library a project relies on, may also make that project vulnerable and will, therefore, be assigned to the latter as well. Naturally, such a flaw can only be fixed in the underlying third-party project and not in the considered project itself, thus making it impossible to link to a fixing-commit.
  Furthermore, does our approach only consider vulnerabilities that have already been fixed, as unresolved ones do, of course, not have a fixing commit.

- **Correctness:** The NVD data is manually curated and validated. Nonetheless, there might be errors in this data, as well as in the one gathered by third-parties. Our approach, however, has some built-in data cleaning techniques. By looking into the actual commits in the source code we ensure that the CVEs included in this thesis do indeed affect this software. CVEs that wrongfully mark the product as vulnerable will not have a corresponding fixing-commit and are automatically discarded. Nevertheless , our automated approach of linking CVEs and fixing commits over common bug identifiers relies on the developers denoting the correct bug in the commit message and also complying with certain syntax

policies. This may lead to errors in the information of about the vulnerability itself but also in the mappings between fixing-commits and CVEs.

- **Bias:** Only including 10 projects in our analysis may introduce a bias in our data towards certain types of software. However, with our set of projects being very diverse we figure that our results will also be applicable to open source software in general.

4.3 Heuristic Lifetime Approximation

Calculating the lifetime of a software vulnerability requires identification of a VCC that introduced the underlying flaw for each fixing-commit, This is a very complex and non-trivial task. Some researchers ([14]) have taken a manual approach to obtain this data. This method is incredibly time-consuming and therefore not suitable for the scale of our work. Furthermore, it may also lead to mistakes in the obtained information. We, therefore, considered automated, heuristic techniques as introduced by *Perl et al.* [15] and *Yang et al.* [17]).

4.3.1 Heuristic Definition

Preliminary evaluation of both algorithms suggested that the Vuldigger Heuristic slightly outperforms VccFinder. Hence, we execute the Vuldigger Heuristic Heuristic ([17]) for every identified fixing commit. This heuristic is defined as follows:

1. Ignore changes in files that do not contain code (We only considered .c, .c++, .cpp, .h, .hpp & .cc file extensions)
2. Ignore comments
3. Ignore empty lines and also ignore whitespace changes by setting the "**-w**" flag with **git blame**
4. For every deletion: blame the deleted line
5. For single insertions: blame the line if it contains one of the following keywords: "if", "else", "goto", "return", "sizeof", "break", "NULL"
6. For every block inserted: Blame the line before and after the block, if it is not a function statement

This leaves us with a list of potential VCCs and their respective number of blames. In prior work the commit that was blamed most often is usually considered as the VCC. The lifetime of the vulnerability could then be calculated as the time span between the fixing-commit and the most-blamed commit. However, our evaluation of these heuristics showed that the correct VCC will only be identified as such in half of all cases. This results in the lifetime of the vulnerability being drastically miscalculated in many cases. But as discussed in 3.1 it is not necessary for us to pin-point an exact commit as VCC. It is sufficient to approximate the point of time the flaw was introduced. Even if the most-blamed commit is not the actual VCC, it is likely to be among the other blamed commits. Thus we take all blamed commits into consideration. Instead of taking the most-blamed commit, we compute the weighted average of the commit date over all blamed commits to estimate this VCC

	Total	Percentage
Correct VCC	369	49.53
Upper bound holds	658	88.32
Lower bound holds	665	89.26

Table 1: Heuristic performance on 721 linux-kernel CVEs

date.
This heuristic date d_h over n commits is defined as

$$d_h = d_{ref} + \frac{1}{\sum_{i=1}^{n} b_i} \sum_{i=1}^{n} b_i (d_i - d_{ref}) \tag{0}$$

with b_i being the number of blames the commit i received and d_i the respective commit date. For easier calculation the dates are represented as difference in days to a static reference date d_{ref}.

Of course, this value will not be correct for all fixing commits, but when considering the average lifetime over multiple vulnerabilities it is a valid representation of the actual lifetime. In addition to the lifetime induced by d_h we also consider the oldest and newest of the blamed commits to provide upper and lower bounds of the lifetime, respectively. The approach of considering the newest commit is similar to the one taken by Li and Paxson, although they used a less evolved version of the heuristic. Nonetheless, this allows us to compare our results to theirs and also investigate by how much their conservative approach underestimated the actual lifetimes.

4.3.2 Perfomance Evaluation of Heuristic

To evaluate the performance of the heuristic and to find empiric evidence for the assumptions made in 4.3.1 we compare the results of our heuristic against a ground-truth dataset ([5]). The Ubuntu Security team has manually curated and evaluated a mapping between CVEs, fixing-commits and the respective VCCs for vulnerabilities in the Linux kernel. The professional experience of the authors and randomized checks of the results assure us that the data is of very high quality.

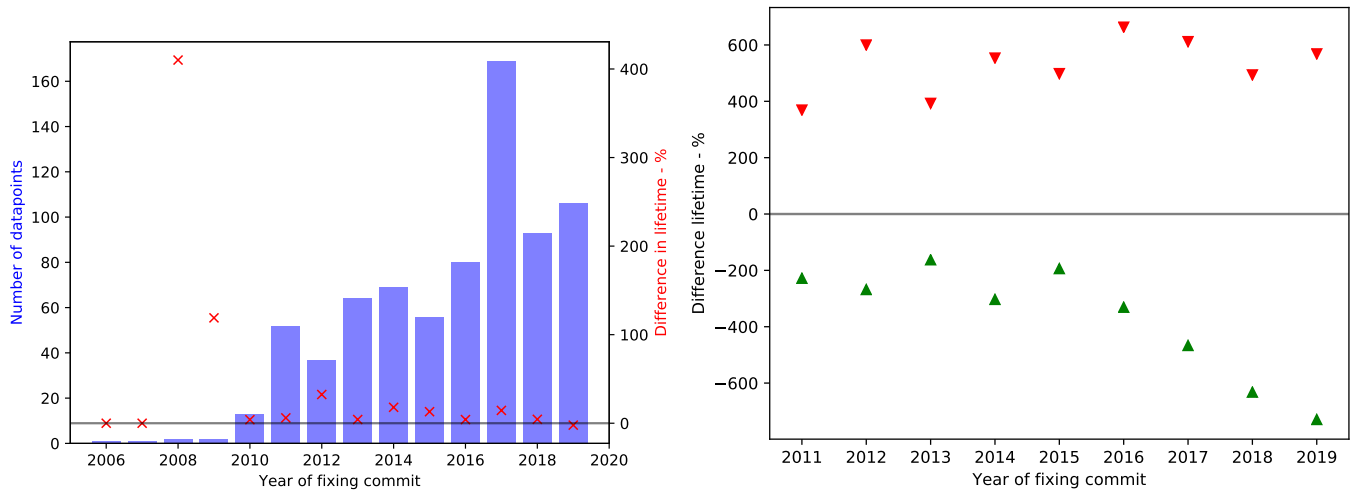
We disregarded a few very large fixing-commits (e.g.: [10, 18]) that mostly only removed drivers and were therefore not representative. We hope that this also limits the bias that may result from considering only the Linux kernel as the evaluation of the heuristic. Furthermore, we also believe this data set to be very representative in itself. With fixing commits ranging from 2006 to 2019 we have data available for over 10 years of development. This allows us to investigate and possibly negate errors in the heuristic that may stem from the increasing age of the repository. Additionally, the Linux kernel is a very complex and diverse project and therefore the vulnerabilities included in our ground truth data set are very diverse as well. The data set consists of 745 vulnerabilities across 48 different Common Weakness Enumerations (CWEs). These CWEs include all kinds of vulnerability types ranging from race conditions and overflows to infinite loops, divisions by zero and other incorrect calculations as well cryptographic issues and improper authentication and privilege management. Consequently, this data set can be considered a good representation for open source software in general. Nonetheless, we were very cautious in our definition and evaluation of our heuristic to avoid over-fitting to this specific project.

	Correct most-blamed		Incorrect most-blamed		Total	
	Mean	Deviation	Mean	Deviation	Mean	Deviation
Weighted	-10.033	375.570	133.154	1322.036	62.233	978.30
Most-blamed	0.0	0.0	209.205	1620.651	105.585	1156.09

Table 2: Difference in days to correct VCC for weighted-average and most-blamed approach

As shown in Table 1 in only half of all cases the heuristic actually identifies the correct VCC. This is nowhere near the accuracy of 95.5 % the authors claimed in their paper. Furthermore, it is also evident that the upper and lower bound is incorrect for about 10% of cases. This validates the assertion that the lifetimes calculated by the heuristics are not suitable for investigating individual vulnerabilities.

To justify the usage of the weighted average (Equation 0) instead of the usual most-blamed approach we compared their performance. Table 2 shows the mean error and deviation in days of both approaches for three different scenarios: For cases were the most-blamed commit was the actual VCC, for CVEs were the VCC was not the most-blamed commit and, lastly, for the entire data set. Naturally, in cases where the most blamed commit was the actual VCC the most-blamed approach does not make any errors. The weighted-average, on the other hand, slightly underestimates the correct lifetime. Nonetheless, this number is still very low. Mostly due to the fact that in many cases only one commit is blamed at all, resulting in the same value for both approaches.
When the Vuldigger heuristic identifies the wrong VCC, however, the weighted average is significantly closer to the correct VCC date than the most-blamed commit. Additionally, the residuals of the weighted average deviate less.
Because of the high error rate of the Vuldigger heuristic, this overall results in the weighted average being more accurate and also less divergent than simply taking the most-blamed commit.



(a) (b)

Figure 1: Error of average approximation wrt. sample size and error of averaged upper and lower bounds

In the later work of this thesis we will investigate the historic trend of vulnerability lifetimes by looking into the average lifetimes of all vulnerabilities fixed in each year. To assess the number of data points necessary for the average to be statistically stable we evaluate the error for each year in the ground truth data set with

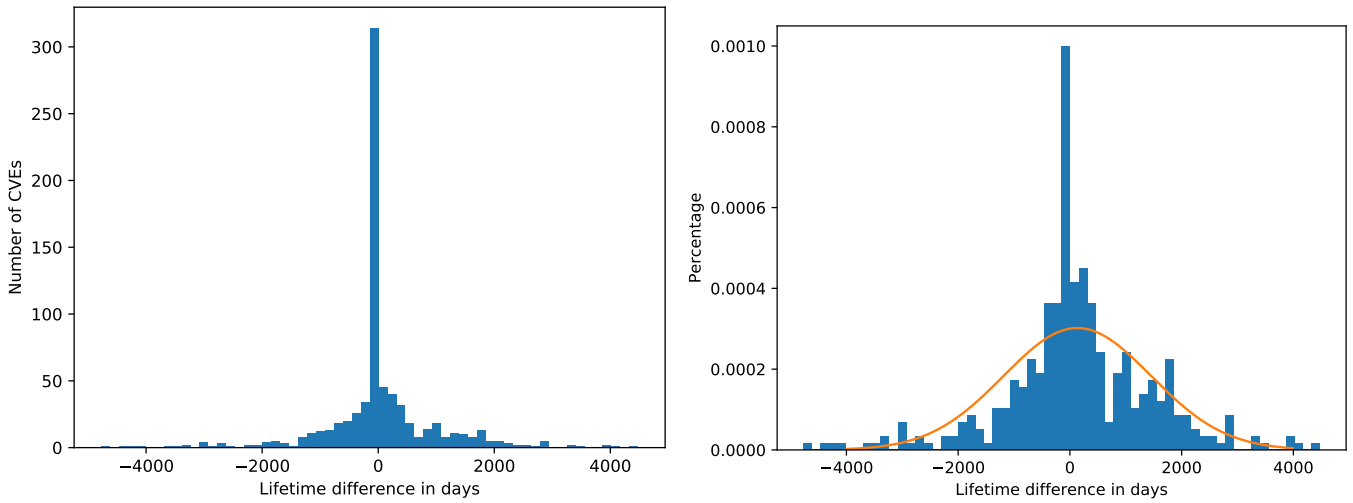
respect to number of vulnerabilities. Figure 1a shows the ratio between the average lifetime approximated by the heuristic and the actual average lifetime grouped by the year of the fixing commit as well as the respective number of data points.
It should be noted that the years 2006 and 2007 only considered a single CVE each. The heuristic blamed the correct VCC in both cases, thus explaining the perfect ratio of 0%. As this is a result of chance we will not consider these two years in our evaluation.
The results of evaluation are two-fold:

1. For low numbers of data-points the heuristic results are prone to heavily miscalculate the average lifetime due to the strong effect outliers may have.

2. For large enough sample sizes the averaged, heuristic lifetime gives a good approximation of the actual lifetime. In this dataset samples with a size of more than 11 datapoints never over- or underestimate the actual lifetime by more than 42.3%. And for sample-sizes of 52 or higher the averaged, heuristic lifetime is never more than 15.7% off.

With the respects to these results we will only evaluate certain aspects of vulnerability lifetimes if we can guarantee at least 20 data points.

With these requirements to the sample size we can now also evaluate if the upper and lower bounds hold. Figure 1b shows the ratio between the difference of the averaged upper and lower bound provided by heuristic and the actual lifetime. For years with at least 20 datapoints our hypotheses clearly holds. The upper bound hereby overestimates the lifetime by 50.3% on average, while the lower bound averages -30.4% underestimation.



(a) (b)

Figure 2: Distribution of heuristic errors for the entire sample and for cases where the most-blamed commit was wrong.

Evaluating the distribution of residuals allows for a more detailed evaluation of the heuristic's behavior. Figure 2 depicts the distribution of the differences between the heuristic lifetime and the correct lifetime. The distribution over the complete sample is shown in Figure 2a. Since the heuristic determines the correct VCC in 50% of cases and many of the fixing-commits result in a small number of blames, most of the heuristic lifetimes are very close to the correct ones. The positive average of 62.23 shows that the heuristic is bound to

overestimate the actual lifetime. A behavior that can also be observed in Figure 1.
Figure 2b only shows the error distribution of CVEs where the heuristic identified the wrong VCC as well as a normal distributed fit. Although, we can say with statistical significance ($p = 1.16e - 05$) that the residuals do not follow a normal distribution, we can clearly see that the errors of a large enough sample size will somewhat average out. This further justifies the usage of the proposed heuristic for statistical analysis.

We thought it might be possible that with increasing age of the repository the heuristic was more likely to blame older commits. This could result in an increasing overestimation of the lifetime for newer fixing commits. Therefore, a discount-factor proportional to the age of the repository would be necessary to compensate for this trend.

Hypothesis 1. *With increasing age of the repository the heuristic is more likely to overestimate the correct lifetime.*

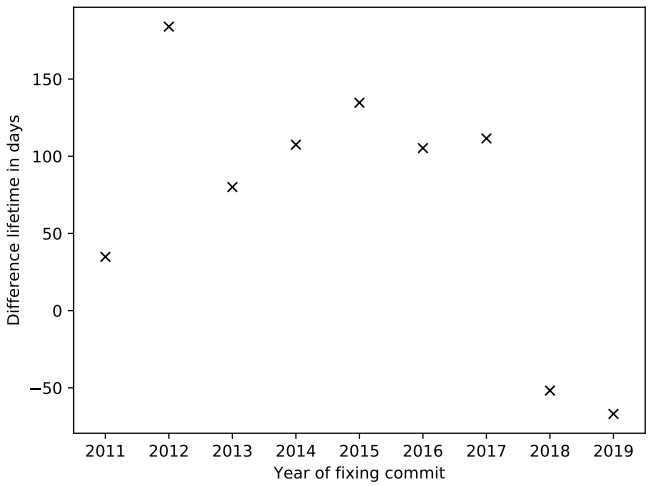


Figure 3: Trend in weighted error

Figure 3 depicts the difference between the averaged heuristic lifetime and the correct lifetime for all the years with at least 20 data points. Since their is no such trend identifiable we reject Hypothesis 1 and do not introduce a discount-factor into our calculations. In fact the Spearman correlation coefficient is even slightly negative thus further justifying the rejection of Hypothesis 1.

4.4 Considered Software Projects

As a starting point for further investigation, we looked into Debian packages with the most CVEs. Debian covers a large portion of OSS and also has a transparent vulnerability disclosure process which allows a detailed investigation of vulnerabilities. In fact, all of the projects we included in our work are distributed under Debian. Nonetheless, these are very popular software products in general that are also being used outside of Debian. While we have been interested in including other open source projects as well, we discovered there were only few or no entries for these products in the NVD. This does not mean that this software is not affected by vulnerabilities but it is, nonetheless, not suitable for this thesis. To get a meaningful insight into

Project	total CVEs	mapped CVEs	fixing commits	suitable CVEs	suitable fixing-commits
Firefox	*1974*	*1386*	*3521*		
Thunderbird	*941*	*683*	*1866*		
Combined	0	1392	3528	1143	2676
Chromium	2320	1123	2099	941	1720
Linux Kernel	3933	1574	2706	1267	1267
MySQL Server	774	0			
Wireshark	576	293	316	279	311
FFmpeg	319	153	183	149	173
Apache HTTP Server	247	206	861	126	329
TCP Dump	160	109		109	111
OpenSSL	207	111	163	100	135
PostgreSQL	120	62	113	53	74

Table 3: Overview of the sample sizes for all considered repositories

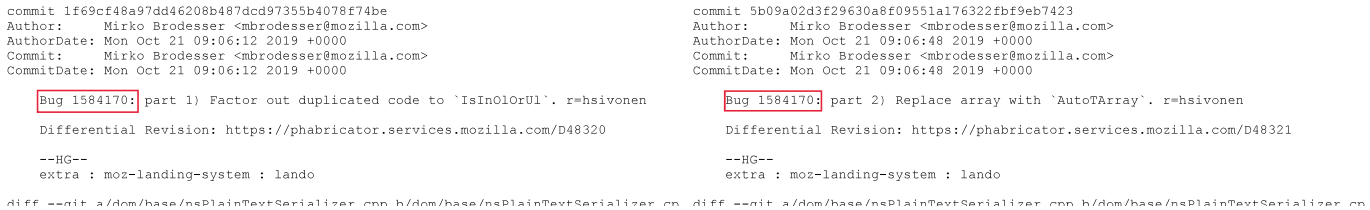
the lifetime of a specific software we only considered cases where there were at least 100 CVEs in the NVD. This ensures that there is no bias towards single vulnerabilities with unusual behavior and also that the our heuristic (4.3.1) produces meaningful results. Additionally, it is necessary that a significant portion of those CVEs can be mapped to a fixing-commit with one of the techniques described in 4.2. The software products meeting these criteria and were included in this study are introduced in this section.

Table 3 provides an overview of all repositories whose lifetimes we evaluate in this work. The first column represents the total number of CVEs in the NVD database and the second one the number of CVEs we could link to a fixing commit with one of the techniques introduced in 4.2. There are multiple reasons a CVE might not be linkable to a fixing-commit. In cases where we had to rely on third party mappings the original authors might not have had the resources to investigate every CVE or they chose to look only at a particular subset for their research. Some CVEs were also introduced in third-party libraries, thus introducing a flaw in the software, but being fixed in a different project. Lastly, the maintenance of vulnerability entries is a manual process. The responsible author might not have had the necessary information regarding a fixing-commit or it was not added to the CVE after the fix had been published.
The third column shows the number of fixing-commits associated with the mapped CVEs. As many CVEs are fixed in multiple commits this number tends to be significantly higher.
The fourth and fifth column are the number of CVEs and respective fixing-commits the heuristic was able to identify a VCC for. These are the number of data-points actually available for our evaluation. Some fixing-commits only make changes to file types we do not consider in the heuristic, which is therefore unable to find a VCC. These changes for example only effect staging or configuration files. In some cases these were even commits only changing text files that document the vulnerability.

For the majority of projects the first three techniques introduced in 4.2 were sufficient to establish a mapping between CVEs and fixing commits. The details regarding the approaches taken for each project, as well as the respective regular expression matchings can be found in the appendix. It should be noted that for some projects this process is also not trivial as they may have multiple different hyperlinks directing to the same bug tracking systems, as well as a different syntax for denoting bug identifiers in commit messages. All

Hyperlink	Resource
http://lists.opensuse.org/opensuse-security-announce/2020-01/msg00000.html	Issue Tracking, Third Party Advisory
http://lists.opensuse.org/opensuse-security-announce/2020-01/msg00001.html	Issue Tracking, Third Party Advisory
https://access.redhat.com/errata/RHSA-2020:0292	
https://access.redhat.com/errata/RHSA-2020:0295	
https://bugzilla.mozilla.org/show_bug.cgi?id=1584170	Permissions Required
https://security.gentoo.org/glsa/202003-02	
https://security.gentoo.org/glsa/202003-10	
https://usn.ubuntu.com/4241-1/	
https://www.mozilla.org/security/advisories/mfsa2019-36/	Vendor Advisory
https://www.mozilla.org/security/advisories/mfsa2019-37/	Vendor Advisory
https://www.mozilla.org/security/advisories/mfsa2019-38/	Vendor Advisory

Figure 4: CVE-2019-17018 References



(a) (b)

Figure 5: Fixing commits

those variations had to be determined through manual examination of NVD entries and commit messages. However, as discussed before this also leads to a more comprehensive data set than simply including a small number of CVEs for smaller projects.

As an example we will discuss the mapping process for one vulnerability effecting Firefox and Thunderbird. CVE-2019-17005 describes a possible buffer overflow during text serialization that could lead to memory corruption and a possible crash. As shown in Figure 4 the references in the NVD contain a link to bug 1584170 on the Mozilla bug tracking website. When mining the repository of the common base code for Firefox and Thunderbird there are two commits that mention the same bug ID in their commit message (see Figure 5). This results in CVE-2019-17005 successfully being linked to its two fixing commits 1f69cf48a97dd46208b487dcd97355b4078f74be and 5b09a02d3f29630a8f09551a176322fbf9eb7423.

As mentioned before we also relied on resources apart from the NVD and the source code itself. For the **Linux kernel** we use the information of the data set introduced in 4.3.2 for all included CVEs. For mapping additional CVEs the introduced techniques are, however, not applicable. The kernel development team around Linus Torvalds does not include bug identifiers in their commit messages and Torvalds has also stated that he does not encourage to link CVEs in fixing commits. Linux distributions like Debian or Ubuntu, on the other hand, often disclose vulnerabilities for their own repositories. The 'Linux Kernel CVEs' project [12] links this information to commits in the upstream Linux kernel project and therefore provides very reliable mappings for CVEs to fixing commits.

Furthermore, the repository of the kernel itself also requires some manual modifications. The current repository on git only goes back to Linux Version 2.6.12, but many of the vulnerabilities covered in this evaluation were introduced before that. To avoid underestimating the actual lifetime we created a complete, historic version of the repository that also includes the git history starting from 0.0.1.

For the **Apache HTTP Server** we use the mappings between CVEs and fixing-commits provided by *Piantadosi et al.* [16]. They automatically generated a set of potential fixing commits from reference information in CVE Entries, commits with the CVE-ID in the commit message and commits that mention related keywords. The authors then manually analyzed these potential fixing commits and created a data set mapping CVEs to fixing commits. As the authors invested manual work in addition to similar methods we used in our data collection process, we are very confident that their data has at least the same quality as ours.
However, none of the commits identified by *Piantadosi et al.* are actually referenced on the current master branch of the official httpd mirror on Github. Nonetheless, we were able to recover that information with a useful feature of the Github API. Although these commits were not accessible on a local clone of the repository, as the branch pointing to them did not exist anymore, Github stores this information in its copy of the repository. By forking the project to a private repository, we were able to create a new branch for each of the commits in question that points to said commit by creating a reference via the Github API [1].

The Debian Security Tracker also collects information regarding CVEs and corresponding fixing commits [7]. While we were able to already obtain most of the mappings listed there with the previously introduced methods, we acquired a few additional links for **Wireshark** and **FFmpeg**.

Table 4 shows other repositories we initially considered for this study but had to discard later. The main reason being that these projects simply did not have enough CVEs in the NVD database to give a significant representation. While this does not mean that the software is free of vulnerabilities it makes it nonetheless unsuitable for our approach.

Repository	NVD Entries
Openjdk	27
MongoDB	18
Nginx	26
CPython	53
libvpx	20
gnutls	18

Table 4: Discarded repositories

4.5 Statistical Analysis

Section about statistical methods and tests being used wrt. to the mathematical theories behind them. Also focus on preconditions of the tests, etc.

5 Analysis & Evaluation

In this chapter we separately investigate and discuss the research questions introduced in 3.2. Each of the subsections is dedicated to one these questions.

5.1 Average Vulnerability Lifetimes

Project	Average Lifetime (days)	Average Lifetime (years)	Lower bound average (days)	Lower bound average (years)
Firefox/Thunderbird	1328.20	3.64	406.79	1.11
Chromium	692.18	1.90	245.61	0.67
Linux Kernel	1761.97	4.82	1186.90	3.20
Wireshark	1814.22	4.97	1046.79	2.87
Apache HTTP Server	1846.34	5.06	771.88	2.11
FFmpeg	1161.79	3.18	784.40	2.15
OpenSSL	2632.19	7.21	1128.89	3.09
TCP Dump	3196.23	8.76	1511.55	4.14
PostgreSQL	2241.83	6.142	760.55	2.08
Overall	***1454.42***	***3.98***	***736.07***	***2.01***

Table 5: Overview of lifetime factors

Table 5 shows the average lifetime computed by our heuristic (4.3.1) for all considered projects. We also provide the lower bound that allows for comparison with the results of Li and Paxson [11].

5.1.1 Results

The lifetimes indicated by the lower bound are similar to the ones of Li and Paxson. The values used in their work tend to be a bit lower, which results from them supplying median values while we work with averages. However, it is very evident that this conservative approach heavily underestimates the actual lifetimes of software vulnerabilities. The average lifetime over our complete data-set is almost twice as high when working with the precise instead of the conservative metric.

These results allow us to find a clear answer to research question 1: On average software vulnerabilities live in the code base for 1454 days which is almost 4 years.

5.1.2 Evaluation

Their are two interesting aspects to these results. First of all there seems to be great difference in lifetime when comparing different projects. The average lifetime of TCP Dump's vulnerabilities is over 4.5 times higher then the average for Chrome. This may have multiple, different reasons. One very simple explanation could be that a some project perform better quality control and internal testing and therefore find vulnerabilities quicker then others.
On the other hand it might also be possible that project like chrome have a very high fluctuation in their code base in general that leads to low vulnerability lifetimes. Because their code gets changed so often and extensively vulnerabilities may not get the chance to become old as they might be resolved unknowingly during regular code changes. Thus resulting in low average lifetimes of vulnerabilities because these recent ones are the only ones in existence.

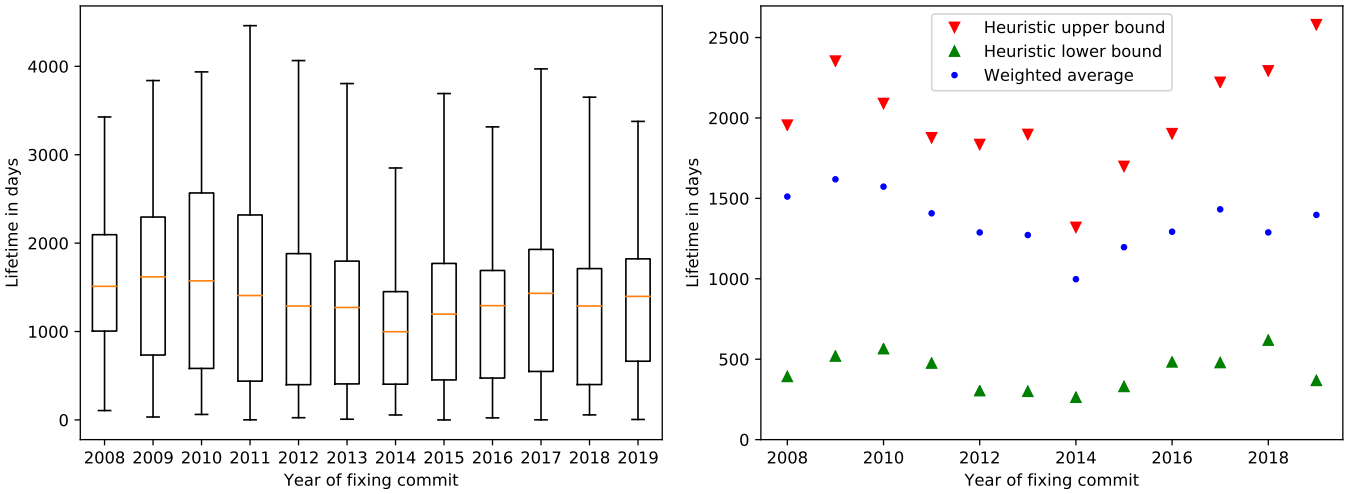
The second aspect is comparing the average lifetimes to earlier research. As stated before *Chou et al.* and *Ozment et al.* computed average lifetimes of 1.8 and 2.6 years for the Linux kernel and OpenBSD in 2001 and 2006. Compared to these results our date would suggest an increase in the average lifetimes of software vulnerabilities. On the other hand are the numbers for the Linux kernel in accordance with the work of *Corbet* and and *Cook* from 2010 and 2016. Both estimated the average lifetime for the kernel to be about 5 years which is also reflected in our results.
We investigate the historic development of vulnerability lifetimes more detailed in section 5.2.

5.2 Historic Development

For answering research question 2 we look into the the development of average vulnerability lifetimes per project. We group the data-set of each project by years of fixing commit to calculate the average age of vulnerabilities fixed in that year. Following we provide two plots for each project. One that shows the average of the vulnerabilities lifetimes as well as the upper and lower bounds as introduced in 4.3.1 per year. Additionally we also show a box-plot with a box extending from the lower to the upper quartile and a line at the mean. The whiskers represent the complete range of the data.

5.2.1 Results

Firefox/Thunderbird



(a) (b)

Figure 7: Firefox - Historic development in average vulnerability lifetime

Figure 7 shows the historic change in vulnerability lifetimes for Firefox/Thunderbird. The results indicate a periodic behavior with average lifetimes ranging from 998 to 1619 days. Overall the trend is slightly decreasing. A simple linear fit indicates an overall decrease of 22 days per year. Figure 7a also shows that the range of data-points follows a similar periodic behavior as the mean itself.

Chrome

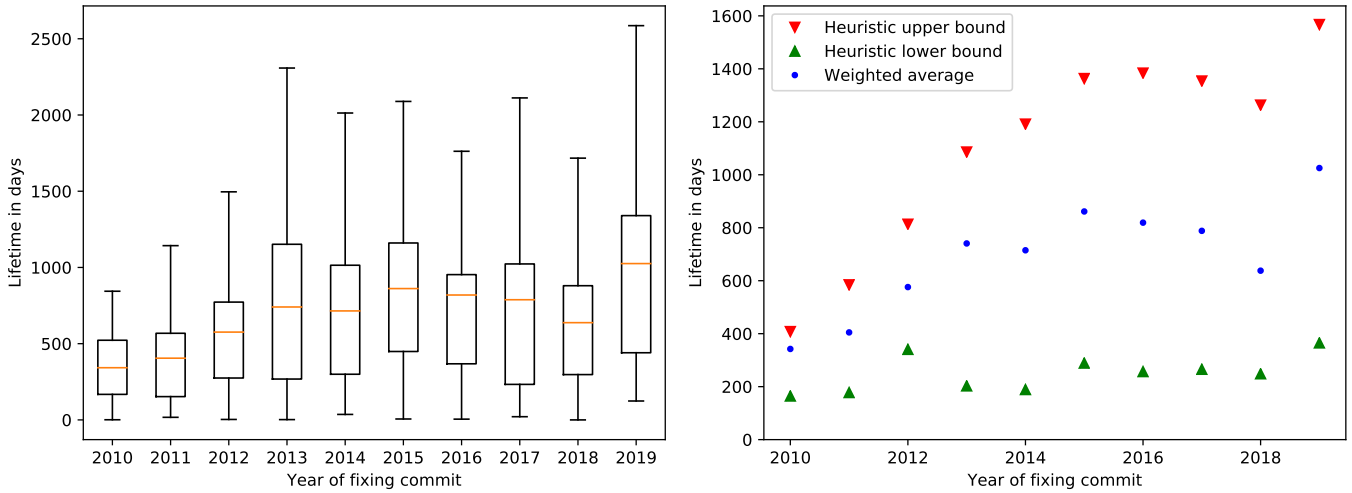


Figure 8: Chromium - Historic development in average vulnerability lifetime

The historic development in vulnerability lifetime for Chromium is depicted in Figure 8. The project also shows some cyclic behavior with the differences in lifetime between two years reaching almost 400 days. In contrast to Firefox however the overall trend is clearly increasing with an average growth of 56 days per year.

Additionally we can observe a linear growth in the average lifetime as well as the range of the data in the first four years of our sample. In that time period the average lifetime increased by almost 140 days per year. The project was initially released in September 2008 and the average for 2010 was 342 days. These results may indicate that a major portion of vulnerabilities fixed in these four years where introduced in the beginning of the development process.

Linux kernel

As depicted in Figure 7b the lifetimes of vulnerabilities in the Linux kernel increase with a linear factor. The delineated fit function indicates a growth of 154 days per year with a correlation coefficient of 0.8921. When disregarding the first year, which can be considered as an outlier, the correlation increases to 0.9606 with a slightly lower growth of 122 days per year.
Furthermore, is the range of lifetimes significantly higher than for the previous projects.

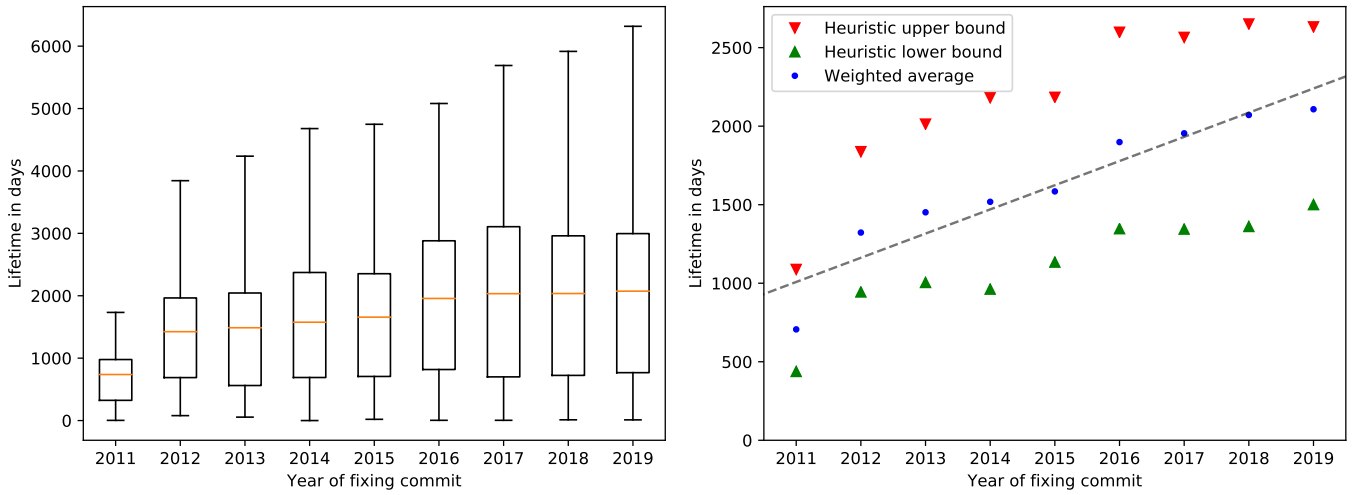


Figure 9: Kernel - Historic development in average vulnerability lifetime with linear fit

Wireshark

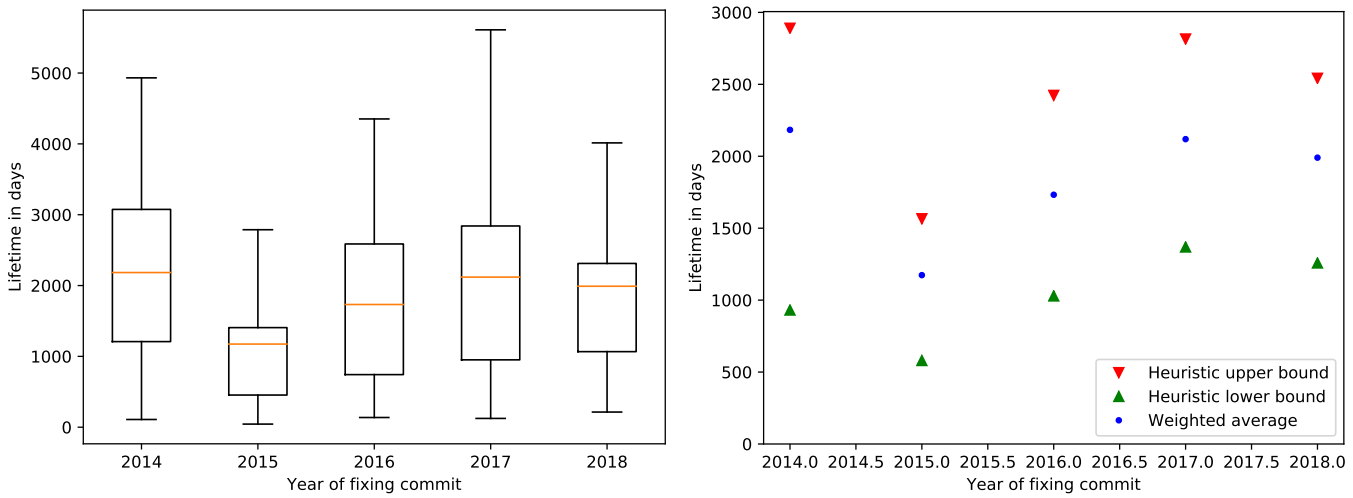
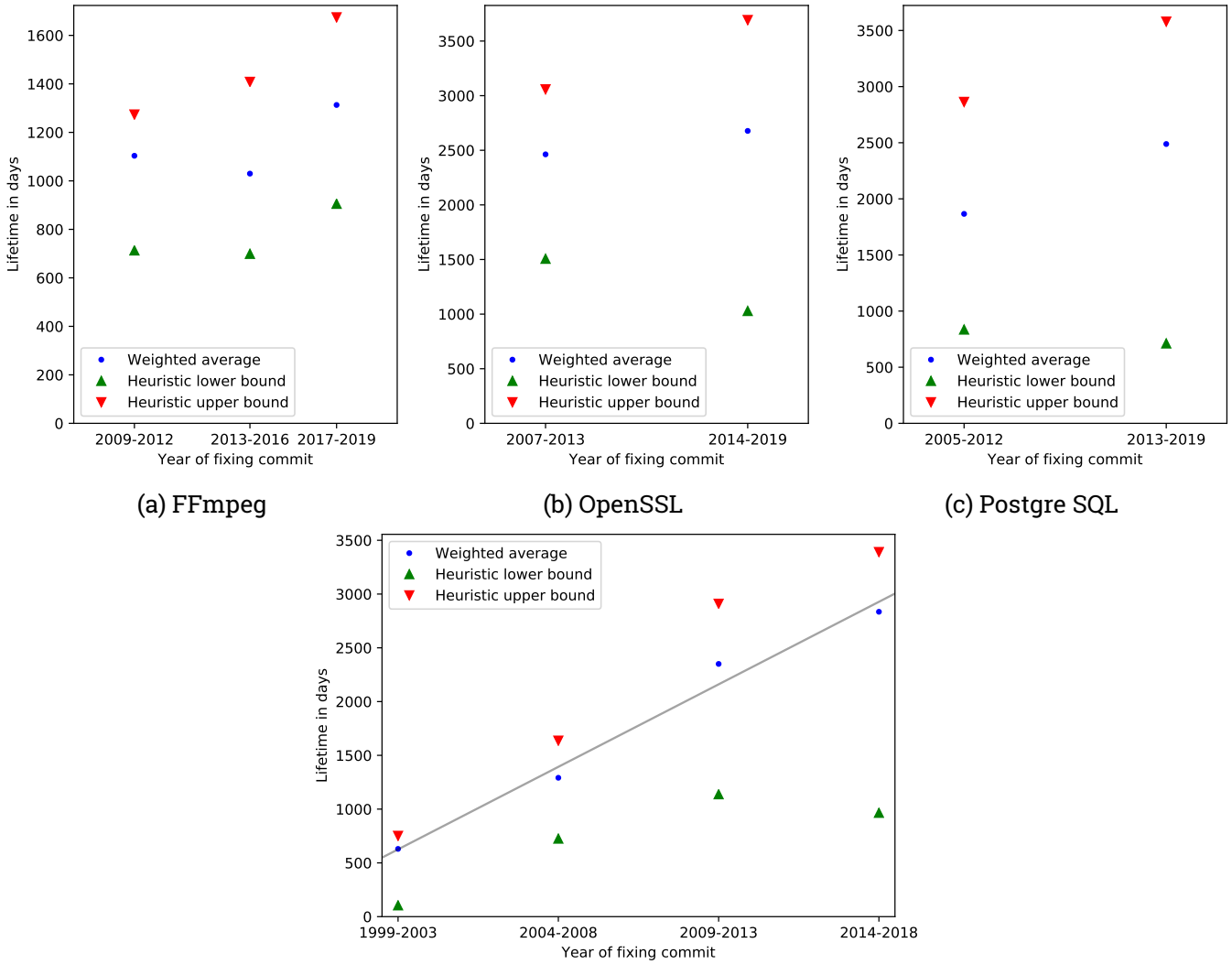


Figure 10: Wireshark - Historic development in average vulnerability lifetime

For Wireshark the available data only allows for an analysis of 5 years as shown in Figure 10. It seems possible that the average lifetimes could follow a periodic behavior. However, the small amount of data on hand only allows to say with certainty that the values fluctuate significantly. The difference between 2014 and 2015 alone is over 1000 days.
Nonetheless, the data indicates an increasing trend in lifetimes within the last 4 years.

Others

The remaining projects introduced in section 4.4 do not have enough CVE entries to be investigated in the same way as the previous ones. To ensure that our requirement of at least 20 data points per aggregated value is met, we do not group the per each year of fixing commit but choose larger time periods. We made sure to choose these time periods to be equally sized. This might lead to only two points in time for our investigation which prohibits us from performing as detailed analysis as before. However, it still provides a trend for the development of vulnerability lifetimes.
Please that due to the amount and structure of the data available for TCPDump it could not be included in this part of the study.



(a) FFmpeg (b) OpenSSL (c) Postgre SQL

(d) Apache HTTP Server

Figure 11: Historic development for other projects

All of these projects show a more or less substantial increase over the years for which data is available.
For the Apache HTTP Server this growth can be explained by a linear function. The depicted fit function indicates an increase of 153 days per year with a correlation coefficient of 0.98.

5.2.2 Evaluation

With the only exception of Firefox all projects show an increase in the average lifetime over the entirety of the available data or at least in the last years. These increases range from an average of 27 days per year for FFmpeg up to 154 days per year for the Linux Kernel. Over all these projects the average increase of vulnerability lifetimes is 65.5 days per year.

The implications of this trend allow for different interpretations. One worrisome explanation can be that software developers introduce more vulnerabilities in their code than they are able to find and fix. This in turn leads increase in the amount of vulnerabilities that remain unfixed with every passing year. Therefore, the age of the vulnerabilities that do get fixed eventually increases over time.
On the other hand, the opposite of that is also a possible explanation for the observed behavior. One might argue that due to new and better tools for code analysis we are able to find older vulnerabilities that were not discovered in earlier inspections. This would mean that more vulnerabilities are closed than being introduced, thus resulting in the average lifetime of fixed vulnerabilities increasing with software age.

5.3 Vulnerability Severity

The severity of software vulnerabilities is estimated by the Common Vulnerability Scoring System (CVSS). Scores range from 0 to 10 with increasing severity. In order to answer RQ 3 we grouped vulnerabilities CVSS scores rounded to whole numbers.

5.3.1 Results

CVSS-Score	Average lifetime
4.0	909.63
6.0	1336.02
9.0	1472.4
10.0	1678.15
7.0	1768.40
5.0	1926.13
2.0	2019.67
3.0	2132.45
8.0	2146.21

Table 6: Average lifetime of vulnerabilities grouped by CVSS Score

Table 6 shows the average lifetime per CVSS score. To analyze if these factors are correlated we formulate the following null-hypothesis:

Hypothesis 2. *There is no monotonic correlation between vulnerability CVSS score and lifetime.*

This Hypothesis can be investigated using a two-sided hypothesis test based on the Spearman rank-order correlation coefficient.
With Spearman's $\rho = -0.023$ we accept the Null-Hypothesis with a confidence of $p = 0.95$.

5.3.2 Evaluation

As discussed before these results were to be expected and the results of *Li and Paxson*. For our data sample the correlation coefficient is even lower than in their evaluation.

5.4 Vulnerability Types

Vulnerabilities are classified using the Common Weakness Enumeration. With respect to RQ 4 we looked into the average lifetimes of the top 10 most common vulnerabilities in our data set. Additionally, we categorized all data using 6 distinct top-level categories:

1. Memory and Resource Management
2. Concurrency
3. Input Validation and Sanitization
4. Code Development Quality
5. Security Measures
6. Others

We chose these categories in a way that ensures every CWE can be assigned to only one of them. Additionally we also believe that these categories give a good representation of the most relevant causes of vulnerabilities while still keeping the number of samples low.
Arguably, memory management and concurrence issues are very closely related. Nonetheless, we chose to keep memory and concurrency issues separated, because they may have very different causes and fixes. Category 4 represents all vulnerabilities related to the negligence of very basic code development concepts. These include divisions by zero, infinite loops, excessive recursion, etc.
The category 5 includes cryptographic issues, as well as flaws related to authentication, permission and privilege management. We also excluded 83 CVEs from this analysis, because had no CWE (CWE-NVD-noinfo) assigned to avoid biasing the results of category 6.

5.4.1 Results

To asses if the CWEs in Table 7 follow different distributions and to verify that the differences are statistically significant we formulate the following null-hypothesis.

Hypothesis 3. *The lifetimes of vulnerabilities for the top 10 CWEs originate from the same distribution.*

As we can not assume normal distribution of residuals we use the Kruskal-Wallis H test. It is a non-parametric version of the One-Way Analysis of Variance (ANOVA) test. Using this test the null-hypothesis can be rejected at a confidence level of $p = 1.11e - 109$.

Accordingly, we also perform the same test for the results over the complete sample shown in Table 8 and introduce a null-hypotheses:

CWE-ID	CWE-Name	Average lifetime	Lower bound
CWE-189	Numeric Errors	748.12	296.45
CWE-20	Improper Input Validation	1380.14	492.31
CWE-399	Resource Management Errors	1605.95	899.68
CWE-264	Permissions, Privileges, and Access Controls	1638.32	884.97
CWE-476	NULL Pointer Dereference	1775.43	1136.78
CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	1824.14	1158.60
CWE-835	Loop with Unreachable Exit Condition ('Infinite Loop')	2076.01	1275.99
CWE-200	Information Exposure	2107.50	1431.27
CWE-416	Use After Free	2211.51	1412.50
CWE-125	Out-of-bounds Read	2548.74	1407.83

Table 7: Average lifetimes of top 10 CWEs

Category	Name	Average lifetime
5	Security Measures	1059.44
3	Input Validation and Sanitization	1393.34
6	Others	1642.87
2	Concurrency	1836.50
1	Memory and Resource	1956.46
4	Code Development Quality	2069.16

Table 8: Average lifetimes per vulnerability category

Hypothesis 4. *The lifetimes of vulnerabilities grouped by categories follow the same distribution.*

Using the Kruskal-Wallis H test this null-hypotheses can also be rejected with a confidence level of $p = 3.24e - 69$. As this test does not indicate which groups differ from one another we also tested the hypothesis for all pair-wise combinations of categories. We used an approximation of a permutation test by limiting the number of permutations investigated to 100.000. For about 75% of all pair-wise comparisons the null-hypothesis can be rejected with a confidence level of under $p = 0.01$.
The exact results of all permutation tests are listed in the Appendix in Table 1.

5.4.2 Evaluation

The results show that different types of vulnerabilities have different vulnerability lifetimes. These differences can also be very high. In our data set, for example, vulnerabilities resulting from lacking quality in code development have almost twice the average lifetime as security related one. On the on hand, this general assessment is in accordance with the results of *Li et al*.
On the other hand, our data their are some significant differences in the rankings of vulnerability types. While the different data sets also lead different CWEs in the top 10, the results are, nonetheless, overlapping. In *Li and Paxson's* work Numeric Errors are at the lower and of the lifetime spectrum, whereas it has the by far lowest average lifetime in our results. This is due to the fact that our data indicates lower lifetimes for numeric errors and also higher lifetimes for other CWEs.

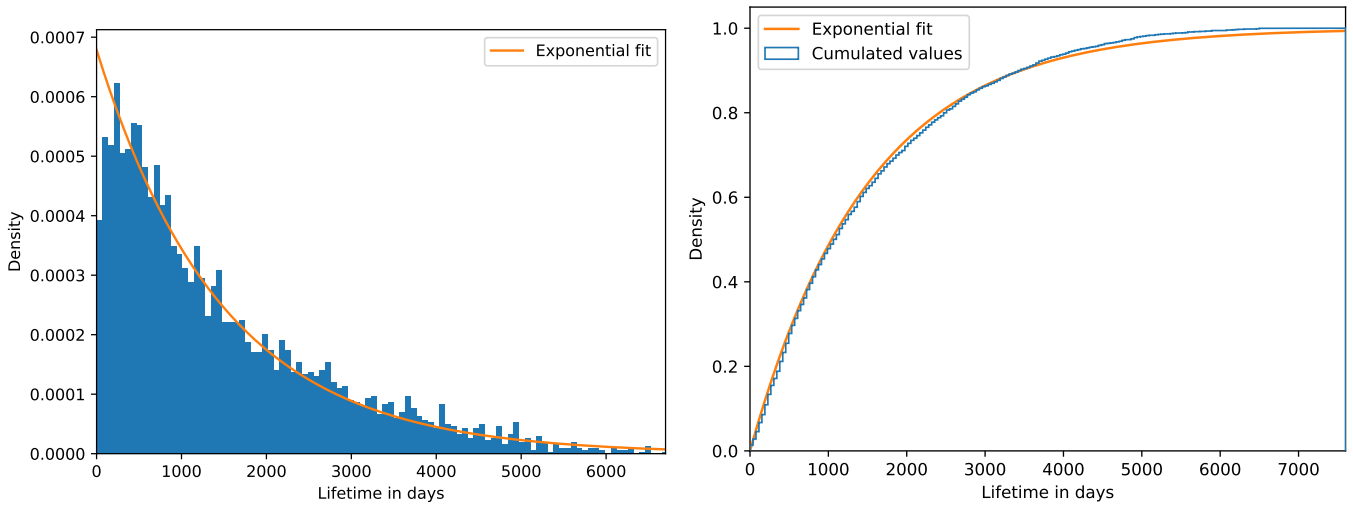
In general are the values put forth by us higher then theirs, even when comparing our lower bound lifetime which is generate with the same approach as their data. This is partially due to the fact of us using averages instead of medians that tend to be slightly lower. However, there still remains a significant difference in the overall results.

The results depcited in 8 indicate that vulnerabilities related to security measurements and input validation and sanitation are fixed quicker than memory or concurrency issues. This may be because the former might be easier to find and locate. A cryptographic issues can often be remedied by simply changing the algorithm used. And a missing input sanitation or privilege check can be resolved with one line of code.
In contrast, memory and especially concurrency issues can be very hard to find. They might only occur within specific executions orders of different tasks and might be part of very complex code parts.

5.5 Vulnerability Distribution

For answering RQ 5 we look for the best possible distribution that fits our lifetime data.

5.5.1 Results



(a) (b)

Figure 12: Distribution of vulnerability lifetimes

We were able to achieve the best results using a scaled exponential distribution. Figure 12a shows a histogram of the all vulnerabilities lifetimes as well the probability density function (PDF) of an exponential fit. The exponential cumulative distribution function (CDF) as well as the cumulative percentage of vulnerabilities with a corresponding lifetime is shown in Figure 12b.
The PDF of this exponential distribution can be described by $f(x) = exp(-x/1473.388)/1473.388$.

5.5.2 Evaluation

The lifetimes of vulnerabilities are clearly exponentially distributed. This is not only applicable to vulnerabilities in general but also the individual software projects. The respective fits and distribution functions of our four largest projects are provided in the Appendix in section A2.

These results have some interesting implications. It means that the majority of vulnerabilities is fixed relatively quick with respect to the average lifetime. With respect to the exponential model we can say that 80% of vulnerabilities are fixed within 32% of the maximum vulnerability age observed. The average is heavily influenced by some long-living outliers with also explains the median always being significantly lower than the mean.

5.6 Effects on Stable Version

In this section we focus on RQ 6. To this end we check which vulnerabilities affected the Debian stable version. Therefore, we considered every vulnerability for which a Debian Security Advisory (DSA) existed as relevant for at least one stable release.
DSA's might only be crafted for more severe vulnerabilities. There could potentially be a number of vulnerabilities that affected the stable release, but where not included in this part of our evaluation. However, we only considered those 1517 CVEs for which a corresponding DSA exists.

5.6.1 Results

	Average lifetime (days)	Lower bound (days)
All CVEs	1454.42	736.07
affecting stable	1745.47	880.55

Table 9: Average lifetimes of vulnerabilities

As shown in Table 9 vulnerabilities that affected Debian stable had a 20% higher average lifetime. This equivalent to ca. 290 days or 9 and a half months. Debian states that stable freeze is usually 7 months long. The observed increase of $9\frac{1}{2}$ months increase may be correlated to the 7 months spend in testing plus $2\frac{1}{2}$ months in testing or earlier development stages.
Apart from the overall increase in lifetime we did not observe any significant changes in other aspects. The distribution of lifetimes can still be described by an exponential function and the historic trend of growing lifetimes is still present.

5.6.2 Evaluation

The results indicate that the Debian *stable freeze* may have a positive effect on its security. Vulnerabilities with low lifetimes may have been already fixed before they are released and could effect customers.
However, defining the *ideal* period of such a freeze is difficult. First of all it depends on the its overall objective. Arguably, the goal of this period is finding bugs in general that would degrade the user experience. The

focus is not necessarily on improving security by actively searching and fixing vulnerabilities.
With how long CVEs remain in the code such periods would to be very long to have a significant impact on the number of vulnerabilities. For example, if we assume that a software features requires 1 year from its initial development to release without any freeze in place. Lets say that after the development and testing the patch would be frozen for an additional 7 months. In our data sample 32% of all vulnerabilities would have been discovered before the release with only 10.5% being fixed in the 212 days of freeze. To achieve an arbitrary goal of 80% of vulnerabilities being found before release the freeze period would have to be over 2000 days long.

Simply delaying the release of a new software patch is not a very suitable mechanism to improve the products security. Also since some bugs that may turn out as vulnerabilities are only found with the feedback of larger number of users. Therefore, even fewer vulnerabilities might be discovered in such long freeze periods.

6 Discussion

7 Conclusion

Bibliography

[1] Git refs. https://developer.github.com/v3/git/refs/#create-a-reference[Accessed: 04.03.2020].

[2] H. K. Browne, W. A. Arbaugh, J. McHugh, and W. L. Fithen. A trend analysis of exploitations. In *2001 IEEE Symposium on Security and Privacy, Oakland, California, USA May 14-16, 2001*, pages 214–229. IEEE Computer Society, 2001.

[3] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. R. Engler. An empirical study of operating system errors. In K. Marzullo and M. Satyanarayanan, editors, *Proceedings of the 18th ACM Symposium on Operating System Principles, SOSP 2001, Chateau Lake Louise, Banff, Alberta, Canada, October 21-24, 2001*, pages 73–88. ACM, 2001.

[4] K. Cook. Security bug lifetime, Oct 2016. https://outflux.net/blog/archives/2016/10/18/security-bug-lifetime/[Accessed: 13.04.2020].

[5] K. Cook. Security bug lifetime, Oct 2016. https://outflux.net/blog/archives/2016/10/18/security-bug-lifetime/[Accessed: 17.02.2020].

[6] J. Corbet. Kernel vulnerabilities: old or new?, Oct 2010. https://lwn.net/Articles/410606[Accessed:13.04.2020].

[7] Debian. Debian security tracker - cve. https://salsa.debian.org/security-tracker-team/security-tracker/-/tree/master/data/CVE[Accessed: 26.03.2020].

[8] A. Dulaunoy. cve-search, Jan 2020. https://github.com/cve-search/cve-search[Accessed 27.02.2020].

[9] N. E. Fenton and S. L. Pfleeger. *Software metrics - a practical and rigorous approach (3. ed.)*. Taylor & Francis Group, LLC, 2015.

[10] gregkh. staging: rtlwifi: delete the staging driver. https://github.com/torvalds/linux/commit/ef4a0c3173736a957d1495e9a706d7e7e3334613[Accessed: 17.02.2020].

[11] F. Li and V. Paxson. A large-scale empirical study of security patches. In B. M. Thuraisingham, D. Evans, T. Malkin, and D. 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.

[12] N. Luedtke. linux kernel cves. https://github.com/nluedtke/linux_kernel_cves[Accessed: 04.03.2020].

[13] U. N. I. of Standards and Technology. National vulnerability database. https://nvd.nist.gov/home[Accessed: 24.03.2020].

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

[15] H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, and Y. Acar. Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In I. Ray, N. Li, and C. Kruegel, editors, *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, pages 426–437. ACM, 2015.

[16] V. Piantadosi, S. Scalabrino, and R. 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.

[17] L. Yang, X. Li, and Y. Yu. Vuldigger: A just-in-time and cost-aware tool for digging vulnerability-contributing changes. In *2017 IEEE Global Communications Conference, GLOBECOM 2017, Singapore, December 4-8, 2017*, pages 1–7. IEEE, 2017.

[18] zx2c4 and gregkh. staging: ozwpan: Remove from tree, Aug 2015. `https://github.com/torvalds/linux/commit/a73e99cb67e7438e5ab0c524ae63a8a27616c839`[Accessed: 17.02.2020].

Appendix

A1 Statistical Test Results

	6	5	4	3	2
1	0.00405	0.00000	0.39139	0.00000	0.49244
2	0.27899	0.00000	0.24050	0.00104	
3	0.00312	0.00000	0.00000		
4	0.00488	0.00488			
5	0.00000				

Table 1: p values of 100.000 sample permutation test. Result from pairwise comparison of vulnerability categories as introduced in 5.4

A2 Distribution Fits

A2.1 Firefox/Thunderbird

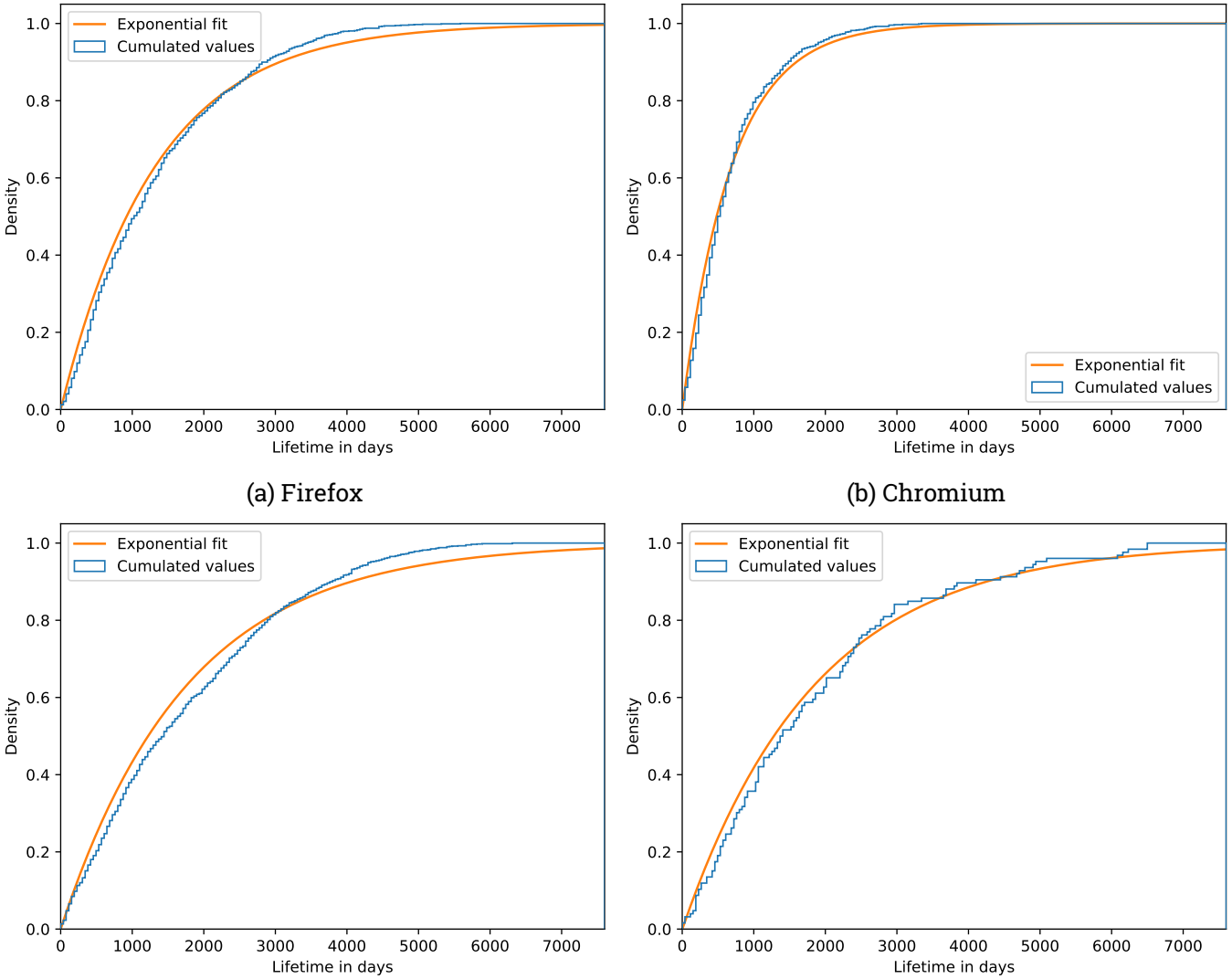
Cumulative fit depicted in Figure 1a.
PDF can be described as $f(x) = exp(-x/1328.204)/1328.204$.

A2.2 Chrome

Cumulative fit depicted in Figure 1b.
PDF can be described as $f(x) = exp(-x/692.175)/692.175$.

A2.3 Linux kernel

Cumulative fit depicted in Figure 1c.
PDF can be described as $f(x) = exp(-x/1761.971)/1761.971$.



(a) Firefox

(b) Chromium

(c) Linux Kernel

(d) Apache Http

Figure 1: Distribution of vulnerability lifetimes per project

A2.4 Apache HTTP Server

Cumulative fit depicted in Figure 1d.
PDF can be described as $f(x) = exp(-(x-2)/1844.341)/1844.341$