NEW APPROACHES TO SOFTWARE SECURITY METRICS AND MEASUREMENTS

NIKOLAOS ALEXOPOULOS

Dissertation
Zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)

genehmigte Dissertationsschrift in englischer Sprache
von Nikolaos Alexopoulos
aus Darmstadt, Deutschland
geboren in Cholargos, Griechenland

Erstreferent: Prof. Dr. Max Mühlhäuser
Korreferent: Prof. Dr.-Ing. Felix Freiling

Tag der Einreichung: 08.02.2022
Tag der Prüfung: 22.03.2022

Fachgebiet Telekooperation
Fachbereich Informatik
Technische Universität Darmstadt
Hochschulkennziffer D-17
Darmstadt, 2022

Nikolaos Alexopoulos:
New Approaches to Software Security Metrics and Measurements

Darmstadt, Technische Universität Darmstadt
Jahr der Veröffentlichung der Dissertation auf TUprints: 2022
URN: urn:nbn:de:tuda-tuprints-215201
Tag der Prüfung: 22.03.2022

SYNOPSIS

Meaningful metrics and methods for measuring software security would greatly improve the security of software ecosystems. Such means would make security an observable attribute, helping users make informed choices and allowing vendors to 'charge' for it—thus, providing strong incentives for more security investment. This dissertation presents three empirical measurement studies introducing new approaches to measuring aspects of software security, focusing on Free/Libre and Open Source Software (FLOSS).

First, to revisit the fundamental question of whether software is maturing over time, we study the vulnerability rate of packages in stable releases of the Debian GNU/Linux software distribution. Measuring the vulnerability rate through the lens of Debian stable: (a) provides a natural time frame to test for maturing behavior, (b) reduces noise and bias in the data (only CVEs with a Debian Security Advisory), and (c) provides a best-case assessment of maturity (as the Debian release cycle is rather conservative). Overall, our results do not support the hypothesis that software in Debian is maturing over time, suggesting that vulnerability finding-and-fixing does not scale and more effort should be invested in significantly reducing the introduction rate of vulnerabilities, e.g. via 'security by design' approaches like memory-safe programming languages.

Second, to gain insights beyond the number of reported vulnerabilities, we study how long vulnerabilities remain in the code of popular FLOSS projects (i.e. their lifetimes). We provide the first, to the best of our knowledge, method for automatically estimating the mean lifetime of a set of vulnerabilities based on information in vulnerability-fixing commits. Using this method, we study the lifetimes of ~6 000 CVEs in 11 popular FLOSS projects. Among a number of findings, we identify two quantities of particular interest for software security metrics: (a) the spread between mean vulnerability lifetime and mean code age at the time of fix, and (b) the rate of change of the aforementioned spread.

Third, to gain insights into the important human aspect of the vulnerability finding process, we study the characteristics of vulnerability reporters for 4 popular FLOSS projects. We provide the first, to the best of our knowledge, method to create a large dataset of vulnerability reporters (>2 000 reporters for >4 500 CVEs) by combining information from a number of publicly available online sources. We proceed to analyze the dataset and identify a number of quantities that, suitably combined, can provide indications regarding the health of a project's vulnerability finding ecosystem.

Overall, we showed that measurement studies carefully designed to target crucial aspects of the software security ecosystem can provide valuable insights and indications regarding the 'quality of security' of software. However, the road to good security metrics is still long. New approaches covering other important aspects of the process are needed, while the approaches introduced in this dissertation should be further developed and improved.

ZUSAMMENFASSUNG

Aussagekräftige Metriken und Methoden zur Messung der Softwaresicherheit können erheblich zur Verbesserung der Sicherheit von Software-Ökosystemen beitragen. Damit kann Sicherheit sichtbar und nachvollziehbar gemacht werden, wodurch weiterhin Nutzer darin unterstützt werden, bei der Auswahl sowie der Interaktion mit Software informierte Entscheidungen zu treffen. Darüber hinaus werden starke Anreize für mehr Investitionen in die Sicherheit von Software geliefert, sodass Software-Anbieter dazu ermächtigt werden, für höhere Sicherheit in ihren Produkten Geld zu verlangen. In dieser Dissertation werden drei empirische Messstudien vorgestellt, die neue Ansätze zur Messung von Aspekten der Softwaresicherheit vorstellen, wobei der Schwerpunkt auf Free/Libre and Open Source Software (FLOSS) liegt.

Erstens: Um die grundlegende Frage zu klären, ob Software im Laufe der Zeit ausgereifter wird, untersuchten wir die Schwachstellenrate von Paketen in stabilen Veröffentlichungen der Debian GNU/Linux Software Distribution. Die Messung der Schwachstellenrate mit dem Fokus auf von Debian Stable: (a) bietet einen natürlichen Zeitrahmen, um die Entwicklung der Ausgereiftheit zu testen, (b) reduziert Rauschen und Verzerrungen in den Daten (nur CVEs mit einem Debian-Sicherheitshinweis) und (c) bietet eine Best-Case-Bewertung der Ausgereiftheit (da der Debian-Veröffentlichungszyklus eher konservativ ist). Insgesamt unterstützen unsere Ergebnisse nicht die Hypothese, dass Software-Sicherheit in Debian im Laufe der Zeit fortschreitet, was darauf hindeutet, dass mehr Anstrengungen unternommen werden sollten, um die Rate von neu eingeführten Schwachstellen signifikant zu reduzieren, z.B. durch 'security by design'-Ansätze wie speichersichere Programmiersprachen.

Zweitens: Um über die Anzahl der gemeldeten Schwachstellen hinaus Erkenntnisse zu gewinnen, untersuchten wir, wie lange Schwachstellen im Code beliebter FLOSS-Projekte (in Bezug auf ihre Lebensdauer) verbleiben. Wir bieten die unseres Wissens nach erste Methode zur automatisierten Schätzung der durchschnittlichen Lebensdauer einer Reihe von Schwachstellen auf der Grundlage von Informationen in Commits zur Entfernung von Schwachstellen. Mit dieser Methode untersuchten wir die Lebensdauer von ~6 000 CVEs in 11 populären FLOSS-Projekten. Unter einer Reihe von Ergebnissen identifizierten wir zwei Größen, die für Software-Sicherheitsmetriken von besonderem Interesse sind: (a) die Spanne zwischen der mittleren Lebensdauer der Schwachstelle und dem mittleren Code-Alter zum Zeitpunkt der Korrektur und (b) die Änderungsrate der zuvor genannten Spanne.

Drittens: Um Einblicke in den wichtigen menschlichen Aspekt des Schwachstellenfindungsprozesses zu erhalten, untersuchten wir die Charakteristika von Schwachstellenmeldern für vier populäre FLOSS-Projekte. Unseres Wissens nach ist dies die erste Methode zur Erstellung eines großen Datensatzes von Schwachstellenberichten (>2 000 Berichte für >4 500 CVEs) durch die Kombination von

Informationen aus einer Reihe öffentlich zugänglicher Online-Quellen. Wir analysierten den Datensatz und ermittelten eine Reihe von Größen, die in geeigneter Kombination Hinweise auf den Zustand des Ökosystems eines Projekts zum Auffinden von Schwachstellen liefern können.

Insgesamt haben wir gezeigt, dass sorgfältig konzipierte Messstudien, die auf wichtige Aspekte des Ökosystems der Softwaresicherheit abzielen, wertvolle Einblicke und Hinweise auf die 'Sicherheitsqualität' von Software liefern können. Der Weg zu guten Sicherheitsmetriken ist jedoch noch weit. Neue Ansätze, die andere wichtige Aspekte des Prozesses abdecken, sind erforderlich, während die in dieser Dissertation vorgestellten Ansätze weiterentwickelt und verbessert werden sollten.

ACKNOWLEDGMENTS

This dissertation would not have been possible without the support of many special people during these almost 6 years of the journey.

First and foremost, I would like to thank my supervisor Max Mühlhäuser. His continuous trust and support, even in challenging times, made this dissertation possible.

Of course a big thank you belongs to the whole TK family who made the last almost-six years of my work-life a pleasure, staying true to the TK spirit. A special thanks goes to my SPIN colleagues – former and current alike. They were always there when I needed them. Without them this dissertation would not have been possible.

I would also like to thank my external collaborators Steffen Schulz and Andy Meneely for interesting discussions and insightful comments that contributed to the results presented in this dissertation. A big thanks also goes to my students who supported me in my research endeavours.

I also owe a huge thank you to my family, and especially my mother, who supported me all the way. The same goes for my friends here in Darmstadt as well as back in Greece.

Finally, I would like to acknowledge the DFG, the BMBF, and the HMWK for funding my research.

CONTENTS

1 INTRODUCTION 1
1.1 Software security, measurement, and the science of security . . . 1
1.1.1 Motivation (Why to measure security?) . . . 1
1.1.2 Why measuring the security of software is hard . . . 2
1.2 State of the Art . . . 4
1.2.1 Common Criteria . . . 4
1.2.2 Vulnerability discovery models and the question of depletion . . . 5
1.2.3 Measuring other attributes . . . 5
1.2.4 Summary . . . 7
1.3 Research Goals . . . 7
1.4 Summary of approaches and contributions . . . 8
1.4.1 General methodology . . . 8
1.4.2 Maturity . . . 9
1.4.3 Lifetimes . . . 9
1.4.4 Effort . . . 10
1.5 Applicability of the developed methods . . . 10
1.6 Outline . . . 10
1.7 Peer-reviewed papers, collaborations and statement over own contributions . . . 11
1.8 Notes on style . . . 12

2 TERMINOLOGY AND BACKGROUND 15
2.1 Terminology . . . 15
2.1.1 Terminology on software metrics . . . 15
2.1.2 Terminology on software vulnerabilities . . . 17
2.2 The vulnerability lifecycle . . . 18
2.3 Data sources . . . 20
2.4 Challenges with vulnerability statistics . . . 21
2.5 Threats to validity in empirical software engineering . . . 23

3 AN EMPIRICAL STUDY ON THE MATURITY OF STABLE RELEASES 25
3.1 Introduction . . . 25
3.2 Motivation and research questions . . . 26
3.3 Specialized Background & terminology . . . 29
3.4 Related Work . . . 30
3.5 Dataset creation methodology . . . 33
3.6 Results . . . 35
3.6.1 Data overview and distribution . . . 35
3.6.2 Vulnerability trends in Debian (H1) . . . 38
3.6.3 Vulnerability Severity and Types (H2) . . . 45

3.6.4 Bug bounty programs (H3) . . . 50
3.6.5 Summary of main findings . . . 56
3.7 Implications and discussion . . . 57
3.8 Threats to validity . . . 60
3.8.1 Threats to construct validity . . . 60
3.8.2 Threats to internal validity . . . 61
3.8.3 Threats to external validity – Generalization . . . 62
3.8.4 Threats to reliability . . . 62
3.9 Conclusion . . . 63

4 AN EMPIRICAL STUDY ON VULNERABILITY LIFETIMES 65
4.1 Introduction . . . 65
4.2 Motivation and research questions . . . 66
4.3 Related work . . . 68
4.4 Vulnerability lifetime in version control systems . . . 71
4.4.1 Defining a vulnerability's lifetime . . . 72
4.5 Dataset creation methodology . . . 73
4.5.1 Mapping CVEs to their VCCs (ground truth) . . . 73
4.5.2 Included projects . . . 74
4.5.3 Linking CVEs to their fixing commits . . . 74
4.6 Lifetime estimation . . . 77
4.6.1 Lifetime estimation in previous work . . . 77
4.6.2 Our approach . . . 78
4.7 Results . . . 82
4.7.1 General . . . 82
4.7.2 Distribution . . . 83
4.7.3 Trends over time . . . 88
4.7.4 Code age . . . 89
4.7.5 Types . . . 90
4.7.6 Case study on impact of fuzzing . . . 93
4.7.7 Summary of main findings . . . 96
4.8 Implications and discussion . . . 97
4.9 Threats to validity . . . 101
4.9.1 Threats to construct validity . . . 101
4.9.2 Threats to internal validity . . . 101
4.9.3 Threats to external validity – Generalization . . . 102
4.9.4 Threats to reliability . . . 102
4.10 Conclusion . . . 102

5 AN EMPIRICAL STUDY ON VULNERABILITY REPORTERS 105
5.1 Introduction . . . 105
5.2 Motivation and research questions . . . 106
5.3 Related Work . . . 108
5.4 Dataset creation methodology . . . 109
5.4.1 Information on included projects . . . 110
5.4.2 Data sources . . . 111

5.4.3 Data cleaning and pre-processing . . . 112
5.5 Results . . . 116
5.5.1 Distribution . . . 116
5.5.2 Temporal characteristics . . . 117
5.5.3 Specialization . . . 120
5.5.4 Motivations . . . 123
5.5.5 Summary of main findings . . . 126
5.6 Implications and discussion . . . 126
5.7 Threats to validity . . . 130
5.7.1 Threats to construct validity . . . 131
5.7.2 Threats to internal validity . . . 131
5.7.3 Threats to external validity – Generalization . . . 131
5.7.4 Threats to reliability . . . 131
5.8 Conclusion . . . 132

6 CONCLUSION 135
6.1 Summary of contributions . . . 135
6.2 Further discussion . . . 137
6.3 Future work . . . 138

A COMPLETE LIST OF OWN PUBLICATIONS 141

B APPENDIX OF CHAPTER 3 145
B.1 Additional Figures . . . 145
B.2 Statistical test results . . . 146

C APPENDIX OF CHAPTER 4 153
C.1 Vulnerability categories . . . 153
C.2 Trends . . . 153
C.3 Lifetime Distribution . . . 154
C.4 Project-specific details on mapping CVEs to their fixing commits 156
C.5 Additional figures . . . 157
C.6 Manual Analysis of Vulnerability-Contributing Commits . . . 160
C.6.1 Introduction . . . 160
C.6.2 Analysis . . . 161
C.6.3 Discussion . . . 165
C.6.4 Conclusion . . . 166

D APPENDIX OF CHAPTER 5 167
D.1 Summary of data sources . . . 167
D.2 Additional figures . . . 168

BIBLIOGRAPHY 171

Figure 2.1 Example tree-like diagram of software attributes with a focus on security quality. 16
Figure 2.2 Simplified plot of a vulnerability's lifecycle. Continuous line shows period of possible exploitation. Events that trigger phase transitions are included. 19
* Figure 3.1 Summary of how the hypotheses investigated in this chapter were derived from the higher level research questions. Produced following an adapted IBIS (Issue Based Information System) notation with the designVUE tool. The main research problem is divided into two sub-problems and the hypotheses H1–H3 are depicted as positions/answers to these sub-problems. 29
Figure 3.2 DVAF's extendable architecture and workflow 34
Figure 3.3 The distribution of vulnerabilities per package (years 2001-2018). Every twentieth package name appears on the x axis for space reasons. The y axis is logarithmic. Packages with at least two vulnerabilities are taken into account. 37
Figure 3.4 A log-log plot (complementary cumulative distribution function) of the distribution of Fig. 3.3. 37
Figure 3.5 Vulnerabilities: distribution and trends. 39
Figure 3.6 Vulnerabilities of php5, during its presence in stable releases, before and after the introduction of the next version (php7) in testing. Vulnerability rate: (a) before the launch of the new version: ≈ 4 vuln./quarter; (b) after the launch of the new version: ≈ 10 vuln./quarter. . . . 40
Figure 3.7 Vulnerabilities of openjdk-7, during its presence in the stable release, before and after the introduction of the next version (openjdk-8) in testing. Vulnerability rate: (a) before the launch of the new version: ≈ 11.3 vuln./quarter; (b) after the launch of the new version: ≈ 10.6 vuln./quarter. 41
Figure 3.8 Vulnerabilities that affected packages of the Wheezy Debian release. *From Q2/2015 to Q2/2016, both Debian 7 (Wheezy) and 8 (Jessie) were supported by the regular security team. This was due to the fact that current Debian practice is that when a new stable version is released, the previous one (now codenamed oldstable) is still supported by the regular security team for another year and then passed to the LTS team. Therefore, the amounts of the regular* period are a higher bound, as some vulnerabilities may have affected only the newer release. We note that in the LTS phase, only one release is supported at a time. 42

Figure 3.9 Vulnerabilities affecting at least two Debian packages. . . 43
* Figure 3.10 Summary of main results of Section 3.6.2 in the form of arguments pro/against the investigated hypothesis. Produced following a relaxed IBIS (Issue Based Information System) notation with the designVUE tool. 46
Figure 3.11 Vulnerabilities severity of the stable release over time. . . 46
Figure 3.12 High severity vulnerabilities of Debian Wheezy. The irregular peak of Q3'17 can be largely attributed to DLA 1097-1 which contained 86 CVEs affecting tcpdump. During the *regular** period 2 releases were concurrently supported by the security team. 47
Figure 3.13 Vulnerability types per year of Debian stable. Labels correspond to root CWE numbers (research view). 48
Figure 3.14 Main vulnerability types of Debian Wheezy, including LTS. 49
* Figure 3.15 Summary of main results of Section 3.6.3 in the form of arguments pro/against the investigated hypothesis. . . . 51
Figure 3.16 Number of claimed bounty reports (left) and new reporters (right) entering the program for IBB (top row) and the HackerOne platform overall (bottom row) over time. 52
Figure 3.17 (5-95%) box plot of USD paid over time for programs in the IBB (top) and all programs in HackerOne (bottom). A further distinction is made between vulnerabilities of any severity (left) and only high/critical (right) severity vulnerabilities. Trend lines for the average (blue) and median (dark red). The only significant OLS trend comes from the top left plot concerning all the bugs reported in the IBB: a statistically significant decrease of the average as well as the median bounty. Detailed statistical test results can be found in Appendix B.2. 53
Figure 3.18 Ratio of bounty amounts (left) and number of reports (right) of IBB reporters (at least 1 IBB report at some point in time) comparing reports in the IBB program against reports for other programs in HackerOne over time. 54
* Figure 3.19 Summary of main results of Section 3.6.4 in the form of arguments pro/against the investigated hypothesis. . . . 55
Figure 4.1 Simplified plot of a vulnerability's lifecycle. Continuous line shows period of possible exploitation. 66
Figure 4.2 Distribution of heuristic errors in days (Excluding datapoints with no error for readability). Equally sized bins. 80
* Figure 4.3 Year trend comparison of ground truth and heuristic data for Linux and Chromium. 82

Figure 4.4 Histograms of lifetime distribution 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. 83
Figure 4.5 Distribution of vulnerability lifetimes. 200 equally sized bins. 84
Figure 4.6 Q-Q Plot comparing the 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. 85
* Figure 4.7 Lifetime distribution per project with theoretical exponential fit (100 equally-sized bins except for the plot for Wireshark which has 50 so as not to have a significant number of empty bins). 86
Figure 4.8 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. Vertical error bars show confidence intervals for each year and "translucent bands around the regression line" give a confidence interval for the regression estimate. All at a 95% significance level and as computed by the seaborn python library via bootstrapping. 87
* Figure 4.9 Comparison of the evolution of the distribution of Chromium and Linux lifetimes over time. 89
Figure 4.10 Distribution fit of lifetimes by year of fix 90
Figure 4.11 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. 91
Figure 4.12 Comparison of lifetime trend in memory related vulnerabilities vs others. 95
Figure 5.1 Distribution of reports per reporter in linear and double logarithmic scales (x axis: reporters ordered by number of reports). 116
Figure 5.2 From top to bottom (all per year): # of CVEs, # of reporters, ratio of reports per reporter (# of reports divided by # of reporters), # of new reporters. Note that the drop for 2020 observed in all plots is due to the time of data collection being in early 2021 (information for some 2020 vulnerabilities may not have been published at that point). 118

* Figure 5.3 Period/duration of engagement for the top 10 and top 20 (human) reporters (time in years between their first and more recent report until now – increased by one), for each project ([5,95] whiskers). Letters in the x axis are the initials of the corresponding projects (Mozilla, Linux, Apache, PHP). The 95% confidence intervals for the median, calculated via bootstrapping (10 000 times), are marked with notches. 120

* Figure 5.4 Number of different CWEs and categories (cats) for each reporter. Third column: for each reporter, what portion of their reports falls into the CWE/category with the most reports (dominant). 121

Figure 5.5 Affiliations associated to CVEs. Internal is for reporters affiliated to the organization behind the project, Other is for reporters affiliated to other organizations, and Unknown is for CVEs which could not be associated to an affiliation. We mark with $ the subsets of the two latter categories that were awarded bug bounties. 124

Figure B.1 Laplace trend tests with 95% significance thresholds (dashed lines). 145

Figure B.2 Bounty amounts in thousands of USD (left) and number of reports (right) of IBB reporters (at least 1 IBB report at some point in time) comparing reports in the IBB program against reports for other programs in HackerOne over time. 145

Figure B.3 The distribution of vulnerabilities in the Debian ecosystem (years 2001-2017), along with the size of the corresponding packages. The scale of axis x is logarithmic. All packages are taken into account. Every tenth package name appears on the y axis for space reasons. 146

Figure C.1 Regular code and vulnerability age for PHP. Vulnerable code seems to be older than regular code for this project. 158

Figure C.2 Regular code and vulnerability age for qemu 158

Figure C.3 Regular code and vulnerability age for ffmpeg 158

Figure C.4 Regular code and vulnerability age for wireshark 159

Figure C.5 Regular code and vulnerability age for openssl. The refactoring process that took place after the Heartbleed bug is evident. A shortage of data points does not allow us to investigate its effect in detail. 159

Figure C.6 Regular code and vulnerability age for postgres 159

Figure D.1 Summary of collected data points and their connections. K stands for the primary (unique) key of the collection. . 167

Figure D.2 Heavy-tailed distribution fits (complementary cumulative distribution function). 168

Figure D.3 For the top 20 (human) reporters for each project: time in years between (a) their first and more recent report until now (last), increased by one to measure period of engagement, (b) their first report and their first peak year, and (c) their first peak year and their last report until now ([5,95] whiskers). Letters in the x axis are the initials of the corresponding projects (Mozilla, Linux, Apache, PHP). 95% confidence intervals for the median, calculated via bootstrapping (10 000 times), are shown as notches. 169

LIST OF TABLES

Table 3.1 The top twenty packages with the most vulnerabilities (counted by CVEs) in time periods (i) 2001-2018 and (ii) 2017-2018. 36
Table 3.2 Most common sets of packages jointly affected by vulnerabilities. 43
Table 3.3 Vulnerability type classification per root CWE number with most dominant examples in our dataset. 48
Table 3.4 IBB dataset summary snapshot on November 2018. . . . 51
* Table 4.1 Overview of selected related work measuring the duration of different phases (also sub-phases and combination of phases) of the vulnerability lifecycle. 69
Table 4.2 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. 76
Table 4.3 Comparison of heuristic performance between the lower-bound approach of Li&Paxson, "our approach 1" based on a repurposed version of the VCCFinder heuristic [88], and "our approach 2" – our optimized heuristic (weighted average). All against ground truth data and measured in days (ME: Mean Error). The last column provides the mean lifetime computed on the ground truth data. . . . 78
Table 4.4 Overview of average lifetimes per project (ordered by number of CVEs) 84
Table 4.5 Vulnerability categories and their mean and median lifetimes (in days) for all CVEs 92
Table 4.6 Vulnerability categories of Chromium, mean and median lifetimes in days 93

Table 4.7 Vulnerability categories for Firefox, mean and median lifetime in days . . . 93
Table 4.8 Vulnerability categories for Linux, mean and median lifetime in days . . . 94
Table 5.1 Breakdown of information sources. PA stands for project-specific advisories, DSA for Debian Security Advisories, USN for Ubuntu Security Notices, RH for the Red Hat Linux BTS, sf for Symantec's securityfocus.com, BTS for the project's bug tracking system, and h1 for HackerOne. The last column is the total unique CVEs that we could obtain any information for (union of all sources). . . . 113
Table 5.2 Breakdown of reporter coverage . . . 115
Table 5.3 Deviations from expected categories. From left to right: reporters with 20 or more reports; out of them, reporters with a deviation w.r.t. categories; out of the latter, reporters with a specific focus category. Note that the values in the second column are a subset of the values of the first, and the values in the third column are a subset of the values in the second. . . . 122
Table 5.4 Reports with bounties per project. Includes a low estimate of confirmed bounties given and a high estimate of reports by a reporter who has received a bounty at least once. . . . 123
Table 5.5 Percentage of vulnerability reporters (excl. reporters marked as 'teams' or 'organizations') with non-cve bug reports and median of such bugs per reporter. . . . 125
Table 5.6 Differences between bugs by vulnerability reporters (reps) and others (rest) as a percentage of the total for each class. All statistically significant ($p < 0.05$ for the Chi-squared test). Severe are bugs with *critical*, *major* or *high* severity (except for PHP for which no severity field exists). Resolved are bugs marked *fixed* in bug reports (except for PHP for which no such field exists, and therefore we considered bugs with associated fixing commits). . . 125
* Table 5.7 Summary of notable parameters identified in this chapter. The section where the relevant results are presented is given in brackets. . . . 133
Table B.1 OLS for reliability trends . . . 146
Table B.2 OLS for bug bounty trends (Fig. 3.17) . . . 148
Table C.1 OLS Regression Results: FFmpeg . . . 154
Table C.2 OLS Regression Results: Httpd . . . 154
Table C.3 OLS Regression Results: Chromium . . . 154
Table C.4 OLS Regression Results: Firefox . . . 154
Table C.5 OLS Regression Results: Linux . . . 154
Table C.6 OLS Regression Results: Wireshark . . . 154
Table C.7 OLS Regression Results: Qemu . . . 154

Table C.8 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% ($p \leqslant 0.01$). 155
Table C.9 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. 156

ACRONYMS

BTS	Bug Tracking System
(C)CDF	(Complementary) Cummulative Distribution Function
CPE	Common Platform Enumeration
CVE	Common Vulnerabilities and Exposures (identifier)
CVSS	Common Vulnerability Scoring System
CWE	Common Weakness Enumeration
DLA	Debian Long-term support security Advisory
DSA	Debian Security Advisory
F(L)OSS	Free(/Libre) and Open Source Software
IBB	Internet Bug Bounty (Program)
IBIS	Issue Based Information System
LTS	Long Term Support
NVD	National Vulnerability Database
OLS	Ordinary Least Squares
VCC	Vulnerability Contributing Commit
VDM	Vulnerability Discovery Model

1 INTRODUCTION

1.1 SOFTWARE SECURITY, MEASUREMENT, AND THE SCIENCE OF SECURITY

Software is nowadays ubiquitous, transcending the boundaries of traditional computing and affecting multiple aspects of everyday life. Securing software systems against malicious actors has thus become a necessity. A common avenue for malicious actors to compromise a system is to find a flaw in its design or implementation that would allow them to make it perform unwanted actions. Such flaws are known as software security bugs or software vulnerabilities[1]. Indeed, a large portion of the work by the computer security community focuses on making software more secure by, e.g. proposing development processes that reduce the probability that vulnerabilities occur in the code, creating tools to detect and fix them, and designing hardware features to mitigate the damage induced by vulnerabilities should they slip through the process. That being said, a large number of vulnerabilities in popular software projects, affecting millions or even billions of users, are discovered and disclosed every day [29].

This thesis investigates the simple, yet elusive (as we explain later) question: *How can we measure how secure a given piece of software is?* Although there exist several useful notions of software security, e.g. speed of patching or operating system security controls, the notion of security that concerns this dissertation is the difficulty of finding new vulnerabilities in a given piece of software. We refer to this notion of security as the "quality of security" of the software.

1.1.1 *Motivation (Why to measure security?)*

After almost two decades of intense research in software security, we still do not have good means to measure the security quality of software[2]. Such metrics would help improve security in many ways:

- First, as a direct consequence, such security metrics could be used to guide decisions on software selection, making security an observable variable in decision-making (see e.g. the recent proposal for "Security Update Labels" [72]).
- Second, and related to the first point, transparent security metrics could transform the market in a way that security rather than being a constant source of annoyance for software vendors, would now be a marketable (and

1 A vulnerability is a more general term than a software security bug since it can include other flaws, e.g. in the processes or people operating software systems. Nevertheless, since in this dissertation we focus on software, when we refer to vulnerability we imply software vulnerability.

2 Refer to Section 2.1.1 for terminology regarding (software) metrics and measurement, as well as information on how our contributions can be expressed with measurement theory notions.

therefore potentially profitable) feature of the software. Overall, this could lead to significantly more security investment and as a result more secure software.

- Third, insights gained from such measurements could provide feedback on the effectiveness of employed tools and processes, and pinpoint weak points that the community needs to focus on. It is difficult to improve what you cannot measure, and thus better software security metrics would help us better choose the most effective tools and processes to apply in each case.
- Fourth, large-scale measurements could shed light on generally recurring patterns, and thus help us build theories on different aspects of the process.

The final point above, although more subtle, is of significant importance. Although finding vulnerabilities is, to a large extent, based on ad-hoc efforts, attaining a deeper understanding of the most influential factors of the process would help the community's strategic planning. This kind of understanding, as a goal, relates to the recently debated need for "more scientific" approaches in the security community [43]. Although "physics-grade" laws for security are unlikely to ever emerge, scientific theories, i.e. observation-driven – yet refutable statements, regarding software security, are possible. For example, theories like *in widely used software projects, each security tester will find on average one new vulnerability per year*, or *memory bugs are, on average, harder to find than other kind of bugs*, would, at the same time provide actionable information regarding long-term planning and shed light on the deeper mechanics of the process (assuming that such theories existed). Like in physics, these statements are scientific in the sense that they are refutable/updateable by new experiments. Also, unlike in physics, such theories can only hope to hold on average, may be temporal, and may leave room for exceptions.

1.1.2 *Why measuring the security of software is hard*

In their 2010 article titled "Why measuring security is hard" [90], Pfleeger and Cunningham pinpoint 9 reasons why information security is in general hard to measure. Among them are: the difficulty to test all security requirements, evolving systems, and the possible inter-dependencies between systems, adversaries and metrics. Such concerns are also reflected in the article on the scientific nature of security by Herley and van Oorschot [43], which was mentioned in the previous paragraph. It is clear that a universal measure (either quantitative or qualitative) of security for a real-world system is presently impossible, and seemingly improbable in the future.

Even when significantly limiting the object of the measurement to a piece of code, measurement does not become much easier. Although many "regular" non-security bugs can come up during the normal operation of a system – and therefore we can analyze their rate of occurrence borrowing tools from reliability theory—security bugs are often different. Instead of a deficiency in functionality, vulnerabilities can be additional unintended features that do not

come up during the normal operation of the system, and are potentially exploitable by an adversary. Pfleeger put it nicely [89]:

> Whereas most requirements say "the system will do this," security requirements add the phrase "and nothing more".

This "nothing more" postscript means that testing for security properties would mean trying out all possible inputs to a system or an application. This is understandably difficult, yet approaches collectively known as formal verification can guarantee that a program satisfies a formal specification of its behavior. However, such approaches generally require a large amount of computational—as well as manual—effort, and are therefore not at this time applicable to most programs. Furthermore, they are not impervious to specification errors, and changes in the functionality of programs might require fresh verification efforts. Formal verification is thus targeted to small critical components. For example, the seL4 [50] microkernel—the first fully formally verified operating system kernel (*functional correctness* property verified starting from an abstract specification down to the C code) – consists of around 10 000 lines of code. In comparison, the Linux kernel weighted around 200 000 lines of code in 1994, and over 27 million at the start of 2020[3]. That being said, although proving full functional correctness is only possible for small components, formal tools can also be used to prove simple attributes of larger codebases, e.g. check for properties like "all array accesses are in bounds" [48], finding bugs in the process.

Since most programs are not formally verified, they contain vulnerabilities, which are gradually discovered and—in the case of many open source projects—publicly disclosed. An attractive approach towards measuring the security of a program would be to just count these faults: the more vulnerabilities discovered for a program, the less secure it is. However, such an approach would not provide much information regarding the security of a program, and may even lead to outright wrong conclusions, especially when used to compare the quality of different programs. Brian Krebs expresses the last point with a fitting metaphor [54]:

> It's a bit like trying [to] gauge the relative quality of different Swiss cheese brands by comparing the number of holes in each: The result offers almost no insight into the quality and integrity of the overall product, and in all likelihood leads to erroneous and—even humorous—conclusions.

A simple way to communicate the fallacies of such an approach is to consider that a home-brewed cryptographic library is unlikely to have any public vulnerability reports—yet it probably is not more secure than its widely used counterparts. People are just not actively trying to find vulnerabilities affecting it.

The above show us that although we can measure a lot of things, it is important to select *what* to measure, carefully. For example, Krebs [54] proposed measuring how many of the vulnerabilities were internally or externally discovered, the amount of time it took for vendors to author patches for known

3 https://www.phoronix.com/scan.php?page=news_item&px=Linux-Git-Stats-EOY2019

vulnerabilities in their products, or the amount of time public exploits existed for such vulnerabilities. Pfleeger and Cunningham [90] are in the same spirit, saying that absolute metrics supporting statements like "Program X is more secure than Y" may be doomed to be flawed, however sets of metrics that support arguments like "the security of X is improving over time" are achievable and useful. In conclusion, as my advisor Max Mühlhäuser put it:

> Measurement is good if and only if we measure the right things.

Therefore choosing what to measure and what information the measurement conveys is a primary challenge in the area. The art of the contributions presented in this dissertation largely concerns this challenge.

1.2 STATE OF THE ART

In this section, we provide a high-level overview of the prior work on software security metrics and measurement studies, in order to highlight gaps in the state of the art. We focus on work on empirical metrics aiming at assessing aspects of the security quality of software, which is most relevant to our work. We provide a more detailed analysis and juxtaposition of the prior work most closely related to the research contributions of each chapter in the "Related work" section of the respective chapter[4]. More general overviews of security metrics (incl. cryptographic strength metrics, system and network security, economic operational security metrics and market metrics) can be found in the Dependability Metrics book by Eusgeld et al. [32], Verendel's critical survey [108] and Pendleton et al.'s survey on system security metrics [86].

1.2.1 *Common Criteria*

The *Common Criteria for Information Technology Security Evaluation*[5] is a framework for computer security certification rather than measurement. On a high level, the Common Criteria provide a detailed framework for evaluating whether or not the processes of specification, implementation and evaluation of a product (commonly referred to as the Target of Evaluation) conform to a set of security requirements (commonly referred to as a Protection Profile). The Common Criteria therefore provide guidelines on what processes may be required (following common sense and "best practices") to achieve a perceived level of security rather than trying to empirically assess the achieved level of security and software quality in the real world.

4 Although vulnerabilities are just a subset of the general class of software defects (bugs), they have been shown to differ in significant ways to other kinds of bugs, mainly regarding their discovery [40] and patching process [58] (it was shown that patches fixing security bugs were significantly different than the rest of the patches), as well as regarding the incentives to find them. Therefore, in this section and for the rest of this dissertation, we focus our related work analysis on works about security issues.

5 https://www.commoncriteriaportal.org/cc/ (ISO/IEC 15408)

1.2.2 *Vulnerability discovery models and the question of depletion*

There have been several works measuring, modeling and analyzing different characteristics of vulnerability discoveries.

A recurring debate is whether or not there is significant reduction in the vulnerability discovery rate during the lifetime of software releases. At the start of the 2000's Rescorla [93, 94] performed an empirical study on vulnerabilities in several popular software projects and observed no significant decreasing trend in the vulnerability discovery rate. He concluded that since vulnerabilities appear infinite when considering the lifetime of a release, vulnerability discovery and public disclosure actually increase the risk of exploitation and are not socially beneficial. The line of work of Ozment [81, 84, 85] came to oppose this statement. In this line of work it was shown that the probability of independent rediscovery of the same vulnerability is not negligible (as Rescorla's model would imply) and therefore finding and patching vulnerabilities is socially beneficial. Ozment and Schechter [84] also showed that there is a decreasing rate in the foundational (present since the first release) vulnerabilities in OpenBSD signaling some improvement in the security of the software.

Apart from the above, there is a long line of work by the Colorado State University [5, 6, 45, 46, 49, 118] on S-shaped Vulnerability Discovery Models (VDMs) that model the number of vulnerability discovery events as a function of time after the release of a software version (some models also use additional information regarding the installed user base over time). Ozment [82, 83] also worked on VDMs, pointing out some limitations of existing models (esp. w.r.t. the dependence between vulnerability discoveries), while Brady et al. [19] introduced a thermodynamics-based VDM[6] that Anderson later employed to show that the security of open and closed source software is theoretically the same [12]. Generally, although several VDMs have been shown to fit empirical data, Massacci and Nguyen [65, 78] showed that none of the considered models (including all of the ones discussed above) are universally better and even the simple model of a constant rate of discovery can be considered the best fit during the first year after a program's release. Ozment had already noted [81, 83] that although models showing depletion can be considered to be a good fit in many cases, models that do not show depletion can also fit the data, and external factors like variations in the effort invested in the vulnerability finding process may be dominating. Overall, the value of sophisticated VDMs has not been clearly showcased and there is no clear answer to the question of whether or not there is vulnerability depletion during the lifetime of a software version.

1.2.3 *Measuring other attributes*

Measuring characteristics of the vulnerability discovery process has bounded potential in measuring notions of software security due to a number of external factors affecting the number of discoveries (e.g. effort as mentioned above).

6 Also known as the BAB Vulnerability Discovery Model (Brady-Anderson-Ball).

Therefore, there exist several studies measuring different aspects of the vulnerability lifecycle. An overview follows.

Speed and quality of patching. Frei [37] and Shahzad [103] performed large scale studies focusing on the timeliness of vulnerability patching in relation to: (a) public disclosure and (b) an exploit being made available. Li and Paxson [58] also looked at similar metrics in addition to quality indicators of applied patches. Overall, the speed and quality of patching can be a measurable indicator of security. Li and Paxson [58] also observed that vulnerabilities live in the code (time between introduction of vulnerability and patch) for long periods of time (more than 3 years based on a lower-bound heuristic estimate). Earlier, Rescorla [94] had roughly estimated (based on available data on affected versions that are known to often contain errors [77, 79]) a median lifetime of 2.5 years for vulnerabilities in the ICAT database (prior format of the NVD) prior to 2003[7], while Ozment [85] had measured a similar median lifetime of a small number of manually analyzed OpenBSD vulnerabilities and commented that such a large lifetime was unexpected given that the code had been reviewed multiple times.

Exploitation risk. The line of work of Dumitras et al. [15, 74, 75, 97] measured indicators of quality in relation to real world attacks captured in the WINE dataset. They measured the relevance of zero day attacks and the amount of time before a vulnerability exploited in a zero day attack is patched. They also measured the speed of patch application in real world systems and how this affects the security of the systems against exploits used in the wild. Such studies provide practical indicators of risk. In another related strand of work, Allodi [8, 9] studied vulnerability and exploit marketplaces and showed a correlation between market activity and real-world exploitation.

Bug bounty programs. There exists a considerable amount of work studying the characteristics of bug bounty platforms [36, 42, 62, 73, 119] Their aim is to measure and propose ways to improve the effectiveness of such platforms. Bug bounty prices offered for a project could be an indicator of quality (reflecting the difficulty of discovering a vulnerability).

The human factor. Although the human factor (esp. vulnerability reporters) has been known to be an influential factor in the vulnerability discovery process, there exists only little work on this issue. Fang and Hafiz [33, 40] performed questionnaire studies to shed light on vulnerability reporters focusing on the practices and tools they follow to search for vulnerabilities and their disclosure processes. No study has yet attempted to capture the effort or scrutiny invested in the vulnerability discovery process.

Code-level metrics. AVA [109] was an adaptive vulnerability analysis approach introduced in the 1990's and adapted from a software failure-tolerance approach (EPA). This approach, which is based on fault injection (injecting vulnerabilities in the code and simulating attacks), can be used to identify regions of the code that may be more vulnerable to an attack of a certain kind and

7 This part of Rescorla's analysis is only available in the original workshop paper that I was able to find online at https://infosecon.net/workshop/downloads/2004/pdf/rescorla.pdf.

can also assess the fault tolerance of software against security threats. Manadhata and Wing [63, 64] introduced an attack surface metric, aiming to measure the 'attackability' of a system. Their attack surface definition and associated algorithm try to capture the amount or ratio of code-level entities reachable by an adversary (from outside the system). Finally, Edwards and Chen [31] performed an empirical study on the relation of the output of a static analysis tool and the discovered exploitable bugs for a number of software projects. They found that the change in the density of issues generated by the source code analyzer can be used to predict the change in the number of exploitable bugs affecting new versions of the same software but cannot be confidently used to compare different software.

1.2.4 *Summary*

It is apparent from the wealth of related work in the area that measuring various aspects of software security is an important pursuit of the research community. However, security metrics and measurement methods are still at an early stage of maturity, compared e.g. to their software reliability counterparts. We are still some way off from having useful ways to measure the security of software. Our work comes to contribute towards achieving this goal by addressing three gaps in the state of the art as detailed in Section 1.3.

1.3 RESEARCH GOALS

The goal of this dissertation is to to explore new paths towards (classes of) metrics and measurement methods (i.e. how to empirically measure these metrics) in order to better measure software security. Based on gaps identified in the state of the art (see Section 1.2), we focus our efforts towards three yet understudied aspects of the vulnerability discovery process that are nevertheless promising to deliver insightful results: *maturity* in stable releases, *lifetimes*, and *effort*. We formulate our goals as three high-level research questions[8]:

1 ***Maturity:* Is the security of software improving over time?** When considering stable releases of software, i.e. releases where only critical patches are applied, one would expect that their vulnerability discovery rate would decrease over time, as the number of vulnerabilities in the code dwindles (as Ozment's observations [82, 83] would suggest). However, no large-scale measurement study investigating this expectation has ever been performed, to the best of our knowledge. Observing patterns of maturing behavior, i.e. gradual decrease of the discovery rate, would indicate that quantitative metrics (e.g. speed of decrease) could be employed to measure the level of maturity of a software release. On the other hand, if no maturity is observable (as Rescorla observed in his work [94]), this would mean that such quantitative metrics are not applicable (at least for the categories of software where this phenomenon

8 Each of the research questions is implicitly followed by the question: *And what can this tell us about the security of a project?*

is observed). Apart from implications in software security measurement, answering this question can have implications regarding the social utility of finding and patching vulnerabilities and on decision-making regarding security investment (balance of investment between bug-finding and secure development).

2 ***Lifetimes:* How long do vulnerabilities live in the code?** Prior work [58, 85], as well as reports in online articles [25, 26] suggest that vulnerabilities remain in the code for large amounts of time before they are discovered and patched. However, no large-scale empirical study measuring vulnerability lifetimes—going beyond rough estimates—exists, to the best of our knowledge. A reason for this knowledge gap is that assessing when a vulnerability was first introduced in the code of a large project is especially challenging. Measuring vulnerability lifetimes, how they vary between projects and over time, how they relate to code quality, and investigating what underlying reasons affect them, would provide valuable insights that could be used to assess the quality of security of a project.

3 ***Effort:* How can we measure human effort in the vulnerability discovery process?** As we stated above, human effort has been identified as a primary factor affecting the vulnerability discovery rate. However, quantities used to indirectly estimate it, like the number of users of a program, are ad-hoc and not clearly connected to the problem at hand (regular users rarely stumble upon security issues). Finding ways to estimate the amount of effort invested in the vulnerability-finding process, and more generally, ways to assess the health of a project's vulnerability-finding ecosystem, would provide valuable indications towards assessing the quality of security of a project.

We approach each of the three research questions above by conducting large-scale empirical studies. A summary of our methodology and contributions follows.

1.4 SUMMARY OF APPROACHES AND CONTRIBUTIONS

In this section, we first go over the general methodology employed for all three of our empirical investigations. Then, we provide a summary of how we applied the methodology in each case.

1.4.1 *General methodology*

In investigating each of the three research questions of Section 1.3, we follow a consistent general methodology:

I. We narrow and specify the scope of the investigation by mapping the higher level research question to a set of specific and direct lower-level research questions.

II. We describe our data source choice and our data collection and cleaning methodology, emphasizing on our objectives of (a) limiting noise and bias

in the analysis and (b) providing open reusable tools and datasets that can be used to reproduce our results and for further research. Regarding the latter, a green box at the introduction of Chapters 3–5 contains information on where to find the code and/or datasets used.

III. When applicable (in the case of vulnerability lifetimes), we provide our approach to extract the quantity of our interest from the collected data.

IV. We investigate the research questions set in the first step of the process (following a data-driven approach).

V. We discuss the implications of the results and the insights attained from the investigation, as well as the threats to the validity of the study.

We proceed to summarize our approach for investigating each of the three main research questions.

1.4.2 *Maturity*

Is the vulnerability rate in popular FLOSS projects decreasing? Is there a decrease in severe vulnerabilities? Does vulnerability type affect the trend? Are bug bounties for FLOSS vulnerabilities increasing?

To investigate such questions, we performed a large-scale study on all software distributed under Debian GNU/Linux. We chose the Debian distribution of the GNU/Linux operating system mainly due to its conservative release cycle, which aims at maximum production-level stability and security for its stable release. We collected a dataset of more than 10 000 vulnerabilities affecting Debian stable releases. To the best of our knowledge, this is the first investigation into the problem that studied a complete distribution of software, spanning multiple versions.

1.4.3 *Lifetimes*

How long do vulnerabilities remain in the code? Are there differences between projects/types of vulnerabilities? How long does it take to find 25/50/75 percent of ultimately discovered vulnerabilities? Are lifetimes increasing or decreasing over time? Are there signs of improved quality?

To investigate such questions, we provided 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 performed the first large-scale measurement of FLOSS vulnerability lifetimes, going beyond the lower bounds prevalent in previous research.

1.4.4 *Effort*

Is dedicated resource allocation or community engagement the primary factor behind vulnerability discoveries? What are the driving factors behind reporting? Are reporters specialized?

To investigate such questions, we performed what is, to the best of our knowledge, the first large-scale empirical study on the people and organizations who report vulnerabilities in popular open source projects. Collecting data from a multitude of publicly available sources (NVD, bug-tracking platforms, vendor advisories, source code repositories), we created a dataset of reporter information for 2 187 unique reporters of 4 677 CVEs affecting the Mozilla suite, Apache httpd, the PHP interpreter, and the Linux kernel.

1.5 APPLICABILITY OF THE DEVELOPED METHODS

Although our investigation for each of the research questions is bounded to a subset of open source projects, our methods can in general be applied to any open source projects with publicly available code repositories and bug information. However, one has to keep in mind that since our main analysis tools are statistical, enough data points should be available for analysis to be meaningful. The methods can also be applied by vendors of proprietary software.

1.6 OUTLINE

The outline of this dissertation is as follows:

- Chapter 1 (current) introduces the motivation and goals of the research performed in this dissertation.
- Chapter 2 covers useful background information.
- Chapter 3 presents a large scale empirical study addressing the first research question of Section 1.3: *Are stable software releases maturing over time?*
- Chapter 4 presents a large scale empirical study addressing the second research question of Section 1.3: *How long do vulnerabilities live in the code?*
- Chapter 5 presents a large scale empirical study addressing the third research question of Section 1.3: *How can we measure human effort in the vulnerability discovery process?*
- Chapter 6 covers the conclusions and outlook of this dissertation.
- Appendices A–D include a complete list of own publications, in addition to supplementary material for each of the three contribution chapters.

1.7 PEER-REVIEWED PAPERS, COLLABORATIONS AND STATEMENT OVER OWN CONTRIBUTIONS

The main content of this dissertation has been published – or is accepted and in the process of being published – at peer-reviewed conferences and journals. Specifically, each of the three main research questions of this dissertation is investigated in a research paper. In the following, I provide more information on those research papers, as well as the collaborations that contributed to the results of this dissertation, also pinpointing my own contributions in relation to contributions by co-authors.

Maturity. The main contributions regarding this research question are presented in:

Nikolaos Alexopoulos, Sheikh Mahbub Habib, Steffen Schulz, and Max Mühlhäuser. 2020. The Tip of the Iceberg: On the Merits of Finding Security Bugs. ACM Trans. Priv. Secur. 24, 1, Article 3 (ACM TOPS).

I was the main contributor of the research and text of this paper. Sheikh Mahbub Habib, Steffen Schulz and Max Mühlhäuser contributed with helpful discussions and insights regarding the main aspects of the work, including comments on earlier versions of the paper.

An early preliminary version of the paper was published as a technical report[9]. Some preliminary related ideas were published as the following poster at CCS 2019:

Nikolaos Alexopoulos, Rolf Egert, Tim Grube, and Max Mühlhäuser. 2019. Poster: Towards Automated Quantitative Analysis and Forecasting of Vulnerability Discoveries in Debian GNU/Linux. 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS 2019).

Lifetimes. The main contributions regarding this research question are presented in:

Nikolaos Alexopoulos, Manuel Brack, Jan Philipp Wagner, Tim Grube, Max Mühlhäuser. How long do vulnerabilities live in the code? A large-scale empirical measurement study on FOSS vulnerability lifetimes. 31st USENIX Security Symposium (USENIX Sec. '22) (to appear).

For this paper, I was the primary contributor of the research planning and specification of the research questions, placement of the work with respect to the state of the art, identification of the impact and implications of the work, as well as the text of the paper. Manuel Brack and Jan Philipp Wagner contributed in the context of their respective bachelor's theses, which I supervised. Decisions regarding data collection and analysis, design choices regarding the employed heuristic, as well as the interpretation of the results, were the product of a collaborative effort of myself, Manuel and Jan, which I led. Manuel Brack contributed the scripts for data collection and statistical analysis of the research

9 https://arxiv.org/abs/1801.05764

questions. Jan Philipp Wagner contributed the implementation of the heuristic for assessing vulnerability lifetimes, as well as the scripts for assessing its performance. Tim Grube and Max Mühlhäuser contributed with helpful discussions and insights regarding the main aspects of the work, including comments on earlier versions of the paper.

Effort. The main contributions regarding this research question are presented in:

Nikolaos Alexopoulos, Andrew Meneely, Dorian-Benedikt Arnouts, Max Mühlhäuser. Who are Vulnerability Reporters? A Large-scale Empirical Study on FLOSS.15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '21).

I was the main contributor of the research and text of this paper. Andrew Meneely and Max Mühlhäuser contributed with helpful discussions and insights regarding the main aspects of the work, including comments on earlier versions of the paper. Dorian-Benedikt Arnouts contributed with the restructuring of the scripts for data collection and analysis, as well as the investigation of research questions regarding differences between non-security bugs reported by vulnerability reporters. His contributions were made during his employment as a student research assistant under my supervision.

The content of the chapters of this dissertation addressing the respective research questions, are modified and extended versions of the contributions presented in the aforementioned research papers. Specifically, differences in the text consist of improvements in the statement of the results, additional results and discussion, as well as changes ensuring the cohesive flow of arguments throughout this dissertation. Considering the fact that I was the main contributor of the text in all aforementioned papers, the majority of text in Chapters 3–5 is transferred, most often with minor edits and suitable restructuring, directly
* from the published papers[10]. New content (extending or replacing content from the published versions of the papers) is marked with an asterisk on the left border at the start of each new paragraph or at the point in a paragraph where new content begins (see asterisk on the left border of this paragraph). New figures and tables are also marked the same way next to their caption and in the table of contents.

During my time as a doctoral researcher, I also published a number of other related papers, the content of which is not included here in order to keep this dissertation more focused. A complete publication list is provided in Appendix A.

1.8 NOTES ON STYLE

In this chapter, I used the personal pronoun "I" to express personal statements and the personal pronoun "we" to refer to the research contributions resulting from collaborative efforts as indicated in the previous section. For the rest

10 with the exception of the first sections of each chapter (Sections 3.1, 4.1, 5.1)

of this dissertation, I will only use the personal pronoun "we" to refer to all contributions.

Regarding digit separators, we use a small space to separate thousands (e.g. 9 999) and a dot (in some cases a comma) as a decimal separator.

2

TERMINOLOGY AND BACKGROUND

This chapter introduces necessary terminology and useful background information.

2.1 TERMINOLOGY

In this section we go over necessary terminology used throughout this document. First, we provide a brief overview of terms associated with software metrics and measurement and how they are used in this document. Then, we go over common terms associated with software vulnerabilities. Readers familiar with vulnerability management practices can skip the second part of this section.

2.1.1 *Terminology on software metrics*

In their seminal textbook on software metrics [35], Fenton and Bieman define *measurement* as follows:

> Measurement is the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way so as to describe them according to clearly defined rules.

They move on to define an *entity* as "an object (such as a person or a room) or an event (such as a journey or the testing phase of a software project) in the real world" and *attribute* as: "a feature or property of an entity". Further, they define a *measure* as: "a number or symbol assigned to an entity by this mapping in order to characterize an attribute [of an entity]". They use the term *metric* both as a synonym for *measure* and also more generally to describe all activities related to measuring attributes of software as in "software metrics".

Interestingly, there is no wide consensus regarding these terms. For example, NIST states[1]: "we use measure for more concrete or objective attributes and metric for more abstract, higher-level, or somewhat subjective attributes"[2]. However, for the rest of this document, we try to adhere to the definitions of Fenton and Bieman as much as possible.

In this dissertation we want to measure aspects of software security. Specifically, we want to measure the attribute *quality of security* (or *security quality*) of a software project, defined as the difficulty of finding new vulnerabilities in a given piece of software. This is a more specific attribute (factor) of the more general attribute "security" of a software project entity (which can in turn be

1 https://www.nist.gov/itl/ssd/software-quality-group/metrics-and-measures
2 Note that here metric/measure are used as subsets of *attribute*.

thought of as a sub-attribute of *dependability*). However, this attribute is still rather high level and abstract. Therefore, we need to measure other factors (related attributes) of the attribute *security quality*. These factors can be, e.g. the "vulnerability discovery rate in stable release" or "the degree of contribution of bug bounties in vulnerability reporting". Empirical metrics/measures characterizing these attributes/factors can be the "number of vulnerabilities per year for stable releases of the software in Debian", or "the ratio of vulnerability reports for which a bounty was awarded". Figure 2.1 provides an example tree-like classification of software attributes with a focus on software quality.

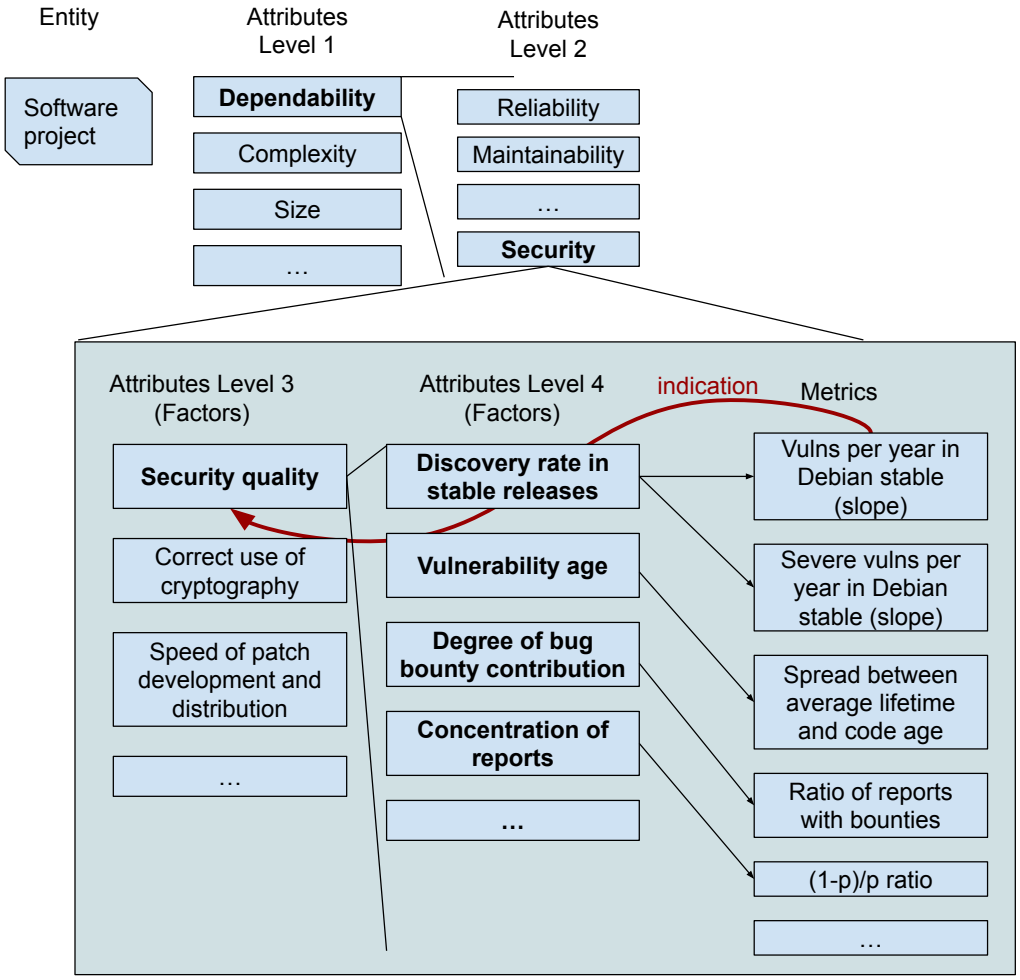


Figure 2.1: Example tree-like diagram of software attributes with a focus on security quality.

The major challenge here is how to interpret the relations of such metrics characterizing factors of *security quality* (level 4 attributes in Fig. 2.1) with the overall attribute *security quality*. Since security is a complex multidimensional process, these relations are also complex and non-linear. Therefore, the best we can do is attain metrics (of such factors/sub-attributes of security quality) that provide *indications* regarding the *security quality* of a software project and, over the length of this dissertation, provide comprehensive discussions regarding

the relations between the measured factors and the "goal" attribute of *security quality*.

2.1.2 *Terminology on software vulnerabilities*

Vulnerabilities and bugs. According to NIST[3] "A *vulnerability* is a weakness in an information system, system security procedures, internal controls, or implementation that could be exploited by a threat source". Thus vulnerabilities include flaws in a system's software design and implementation but also other factors, e.g. issues having to do with hardware or with the personnel controlling a system. However, for the rest of this document, the term vulnerability will be used to refer to software vulnerabilities, i.e. software security bugs. A *software security bug* is a flaw in a computer program, caused by erroneous, missing, or superfluous code, that causes the program to behave in ways that could be exploitable by a threat source.

CVE. The Common Vulnerabilities and Exposures (CVE) program is an international community-driven effort overseen by the MITRE corporation[4]. Its mission is to "to identify, define, and catalog publicly disclosed cybersecurity vulnerabilities"[5]. According to MITRE's definitions [6]:

> A CVE ID is a unique, alphanumeric identifier assigned by the CVE Program. Each identifier references a specific vulnerability. A CVE ID enables automation and multiple parties to discuss, share, and correlate information about a specific vulnerability, knowing they are referring to the same thing.

In the scope of this dissertation, and following common practice in related work, when we refer to a CVE, we refer to the vulnerability with the respective CVE ID. The assignment of CVE IDs to vulnerabilities is performed by CVE Numbering Authoriries (CNAs). According to MITRE:

> [A CNA is] An organization responsible for the regular assignment of CVE IDs to vulnerabilities, and for creating and publishing information about the Vulnerability in the associated CVE Record. Each CNA has a specific Scope of responsibility for vulnerability identification and publishing.

A CVE Record contains descriptive data about the vulnerability associated with a CVE (ID), usually consisting of a short description of the vulnerability and links to external sources (e.g. vendor security advisories).

CVSS. The Common Vulnerability Scoring System (CVSS) [66] is a popular open specification for assessing the relative severity of vulnerabilities. Analysts can assign individual scores for different contributing factors of a vulnerability's severity (e.g. exploitability, impact), and a formula combines these scores

3 https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-30r1.pdf
4 https://cve.mitre.org/
5 https://cve.mitre.org/about/ (04/03/2021)
6 https://cve.mitre.org/about/terminology.html (04/03/2021)

into a base score for a vulnerability in the range $[0.0, 10.0]$. When insufficient data are present for analysts to assign a score, a worst-case approach is followed, e.g. when no data are available, a vulnerability will receive a score of 10.0. Qualitative severity ranking of "Low", "Medium", "High" are also common, e.g. the NVD (see Section 5.4.2) uses the ranges $[0.0 - 3.9]$, $[4.0 - 6.9]$, and $[7.0 - 10.0]$ of the CVSS (v2.0) base score, for this purpose. For more information regarding CVSS, refer to citation at the start of this paragraph.

CWE. According to MITRE[7]: "Common Weakness Enumeration (CWE) is a community-developed list of common software and hardware weakness types that have security ramifications". Weakness types can be hierarchically organized in tree structures. The reasoning behind a hierarchical organization (which CWE types are relevant and how they are related) are expressed by a *View*. For example:

> [The *Research Concepts* View] is intended to facilitate research into weaknesses, including their inter-dependencies, and can be leveraged to systematically identify theoretical gaps within CWE. It is mainly organized according to abstractions of behaviors instead of how they can be detected, where they appear in code, and when they are introduced in the development life cycle.[8]

For the purpose of this dissertation, we use the Research Concepts hierarchical organization of CWE types to organize vulnerabilities where necessary. To give an example, in this view, a typical vulnerability type "CWE-416: Use After Free" is a child type of "CWE-825: Expired Pointer Dereference", which is in turn a child type of both "CWE-672: Operation on a Resource after Expiration or Release" and "CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer". Continuing to climb the tree, CWE-119 is a child of "CWE-118: Incorrect Access of Indexable Resource ('Range Error')", and CWE-118 is a child of "CWE-664: Improper Control of a Resource Through its Lifetime", which is a top-level type (most general category) for the Research Concepts View.

Bug bounty platforms. Bug bounty platforms work in coordination with organizations to offer financial rewards (*bounties*) for the reporting of vulnerabilities affecting the latter's products and online services. Notable pioneers include HackerOne[9] and Bugcrowd[10]. These platforms have succeeded in attracting a large number of skilled hackers to their programs and have paid out large amounts of bounties (e.g. more than 100 million USD for HackerOne[11]).

2.2 THE VULNERABILITY LIFECYCLE

The concept of a vulnerability's lifecycle, as introduced by Schneier [100] and Arbaugh et al. [13] and further enhanced and discussed in a number of sub-

7 https://cwe.mitre.org/about/index.html
8 https://cwe.mitre.org/data/definitions/1000.html
9 https://www.hackerone.com/
10 https://www.bugcrowd.com/
11 https://www.hackerone.com/sites/default/files/2018-03/HackerOne_Press_Kit.pdf

sequent works [15, 37, 85], describes the succession of phases a vulnerability goes through, starting with its introduction in a codebase, and ending when all affected machines have been patched. In Figure 2.2, we provide a rather simple version of the vulnerability lifecycle, fitting our needs. We proceed to provide details on the events that define the phase boundaries.

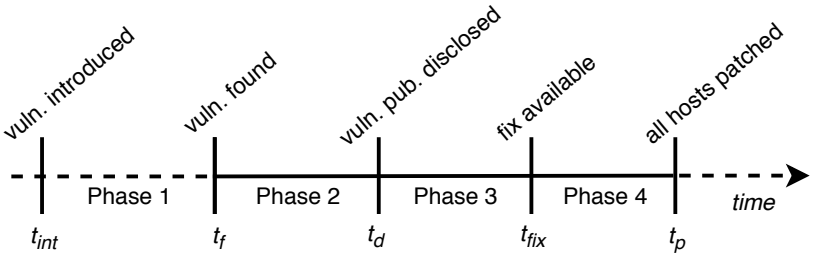


Figure 2.2: Simplified plot of a vulnerability's lifecycle. Continuous line shows period of possible exploitation. Events that trigger phase transitions are included.

Phase 1: The life of a vulnerability starts with the introduction of vulnerable code in the codebase/repository of a program. This event occurs at *Introduction Time* t_{int} (in other literature also called *Injection Date* [85] or *time of vulnerability creation* [37]). Another milestone in Phase 1 is defined as a vulnerability's *Release Date* by Ozment [85] as "the date of public release for the system that first contains the vulnerability". In FLOSS projects with public development repositories, the *Introduction* and *Release* dates can be thought to coincide, since anyone can download and build the latest development version of a program immediately after the change introducing the vulnerability. A vulnerability in this phase is not exploitable, since its existence is not known to anyone.

Phase 2: The second phase of the lifecycle starts when someone finds the vulnerability (at time t_f in Figure 2.2). This event is commonly known as vulnerability *discovery*. After this point, and until the end of its life, a vulnerability is exploitable. Another event that may occur in this phase is the *Private Disclosure* of the vulnerability to the software vendor or security team. The disclosure can be either due to a report by the initial discoverer (e.g. when the initial discoverer is an ethical/white hat hacker or internal to the organization), or due to a report by someone who re-discovered the vulnerability. Ozment refers to this event as just "Disclosure", while Bilge and Dumitras [15] refer to this event as "Vulnerability discovered by the vendor". Whether or not exploitation occurs in this phase depends on several factors, including the intentions of the discoverer, and the difficulty of implementing an exploit. Attacks exploiting vulnerabilities in this phase are called *zero day attacks*[12].

Phase 3: The third phase begins with the public disclosure of the vulnerability (at time t_d). The disclosure can be either through official security advisories, or

12 Although some definitions of *zero day attacks* require that the vulnerability is unknown to the vendor, i.e. the attack happens before the *Private Disclosure* event, we adopt the definition of Bilge and Dumitras [15] that classify all attacks prior to the public disclosure of a vulnerability as zero day attacks

public fora and mailing lists. During this phase, attackers have knowledge of the vulnerability, while defenders have no patch available to protect themselves. Due to the apparent danger of this asymmetry, common disclosure practices dictate that a vulnerability should never enter this phase, i.e. it should transition from Phase 2 to Phase 4 directly.

Phase 4: The fourth and final phase of a vulnerability's lifecycle begins when a fix, i.e. patch, correcting the vulnerability is first publicly available, e.g. via a commit in the development repository (at time t_{fix}). Ozment refers to this point in time as "Patch Date". Phase 4 is dedicated to the distribution and application of the patch to the end hosts. The life of a vulnerability ends when all vulnerable hosts are patched (at time t_p). The same vulnerability can be resurrected when a host installs an unpatched version of the software. The vulnerability will enter Phase 4 again until the host is patched.

The order of occurrence of the events defined above is not strict and many complexities may appear. In addition, some events may never occur. For example, a vulnerability may never be discovered, let alone be publicly disclosed. Also, the fix can be made available and the patching process may commence prior to the public disclosure of a vulnerability. Interestingly, the fix may become publicly available via a code commit before the public disclosure of the vulnerability, providing attackers who monitor code commits with an advantage [58]. Overall, the concept of a vulnerability's lifecycle, as presented in this section, serves the purpose of familiarizing readers with the terminology, processes and stakeholders related to a vulnerability, rather than providing a strict timeline of events.

2.3 DATA SOURCES

In general, data-driven studies rely on the existence of good-quality datasets. Specifically, to investigate the three research questions outlined in Section 1.3, we require information regarding the existence of a vulnerability, its severity and type, information about its fix, information about who reported it, etc.

For each of the three research questions, we construct suitable datasets by combining information from a number of sources. Although the specifics regarding the construction of the datasets for each of our three empirical studies differ (and will be described in the dedicated chapters), here we provide some general background about the sources we used. Note that our research focuses on Free(/Libre) and Open Source Software (F(L)OSS).

The NVD. The National Vulnerability Database[13] (NVD) keeps track of all vulnerabilities which have been assigned a CVE ID by MITRE (see 2.1). It also includes information regarding a vulnerability's severity (CVSS score), and type (CWE), along with links to external sources.

Vendor advisories. Software vendors and distributors publish security advisories regarding security vulnerabilities. Examples include the *Mozilla Founda-*

13 https://nvd.nist.gov/

tion Security Advisories[14] and the *Debian Security Advisories*[15]. These advisories often provide additional information (compared to the NVD) for vulnerabilities assigned a CVE ID.

Bug tracking platforms. Some projects employ bug tracking platforms to track bugs. Some of those projects, like Mozilla Firefox[16] and Chromium[17] use their bug tracking platforms to track vulnerabilities as well. These resources often provide additional useful information regarding a vulnerability, including details about how the issue was discovered and patched. These resources can also provide ways to identify the commit that fixed a vulnerability in a project's repository, e.g. by a mention of the commit in the bug report or by the inclusion of the bug identifier in the commit message.

Third-party databases and mailing lists. Further information about vulnerabilities is exchanged and collected in third party mailing lists and databases, e.g. the *BugTraq* mailing list[18] and the associated *SecurityFocus Vulnerability Database*. Specifically, the latter asset is useful for us as it collects information on people credited with the discovery of a vulnerability.

Bug bounty platforms The HackerOne bug bounty program is one of the pioneers in the area and also covers open source software in the context of its *Internet Bug Bounty* program[19]. Information regarding the people who discovered the vulnerabilities and the financial rewards given can be mined from the publicly visible portions of the platform.

Code repositories. All of the FLOSS projects that we use in our investigations use a "commit-based" version control system for development (git, svn or mercurial). Obviously such repositories are useful since they provide access to the codebase of the project. They also provide potentially useful additional information via commit messages and metadata, such as the committer and the date of a commit.

2.4 CHALLENGES WITH VULNERABILITY STATISTICS

Issues that threaten the validity of results stemming from statistical analysis of data from vulnerability databases are nicely presented by Christey and Martin [22][20]. They present four types of bias that affect the quality of vulnerability databases, such as the NVD:

- *Selection bias.* Selection bias affects gathered data in two ways. First, researchers may focus their efforts in finding vulnerabilities in specific projects, only

14 https://www.mozilla.org/en-US/security/advisories/
15 https://www.debian.org/security/
16 https://bugzilla.mozilla.org/home
17 https://bugs.chromium.org/p/chromium/issues/list
18 https://www.securityfocus.com/archive (note: since the acquisition of SecurityFocus assets by Accenture Security from its previous owner Symantec, it is unclear whether the mailing list will continue operating although Accenture Security has indicated that this will be the case https://www.securityfocus.com/archive/1/542248.)
19 https://www.hackerone.com/internet-bug-bounty
20 Quoted text in this section is taken from this resource.

look for vulnerabilities of a specific type (e.g. buffer overflows), or only use specific tools. This affects which vulnerabilities are discovered. If nobody is looking for vulnerabilities in a product, none will be discovered. Second, collection of information about vulnerabilities in a vulnerability database is a best effort process. No such database can be considered complete, i.e. containing all information that has been published in the internet (in all languages) about all products. People supporting vulnerability databases make decisions on what products to cover and which sources to prioritize. A specific manifestation of selection bias pinpointed by Ozment [85] is *vulnerability discovery dependence*, i.e. selection bias may lead to a large number of vulnerabilities with some common characteristics to be discovered (and subsequently published) in a short period of time. For example, a vulnerability researcher may decide to look for a specific type of vulnerabilities on a specific product for a few days. Vulnerability discovery dependence has a significant impact on measurements of the time between discovery events (e.g. in Vulnerability Discovery Models).

- *Publication bias.* "Publication bias governs what portion of the research gets published. This ranges from 'none', to sparse information, to incredible technical detail about every finding." Publication affects researchers, vendors, as well as the vulnerability database maintainers. Each of those parties may have reasons not to publish a discovered vulnerability.

- *Abstraction bias.* Abstraction bias is a term crafted by Christey and Martin "to explain the process that VDBs [vulnerability databases] use to assign identifiers to vulnerabilities". The problem here lies in differing conventions regarding what counts as one vulnerability (a unique identifier in the database). "Depending on the purpose and stated goal of the VDB [vulnerability database], the same 10 vulnerabilities may be given a single identifier by one database, and 10 identifiers by a different one". Different abstraction levels may also be used by vendors when disclosing issues, or by researchers when reporting issues. The CVE database operates at an intermediate "coordination" level of abstraction, e.g. one CVE may correspond to more than one Bug IDs of a project's bug tracking system. Overall, data analysis should be performed at a consistent level of abstraction.

- *Measurement bias.* "Measurement bias refers to potential errors in how a vulnerability is analyzed, verified, and catalogued". Regarding a vulnerability database, this type of bias refers to any errors in vulnerability entries (e.g. wrong affected versions, wrong CVSS score) or incomplete information. Such errors can influence vulnerability statistics in seemingly unexpected ways. For example, CVE entries with no detailed information are assigned a maximum CVSS score on a worst-case assumption.

Although there is no way to completely eliminate bias in vulnerability studies, researchers should acknowledge, try to mitigate, and discuss the effect of different sources of bias on the measurement results. We discuss how we achieve this in a systematic way in the following section.

2.5 THREATS TO VALIDITY IN EMPIRICAL SOFTWARE ENGINEERING

In this dissertation we follow the systematic approach proposed by Runeson and Höst [96] in their seminal work "*Guidelines for conducting and reporting case study research in software engineering*" to report on threats to the validity of our results. Runeson and Höst propose a classification consisting of four aspects of threats and validity-related concerns, that we adopt:

- **Threats to construct validity.** This aspect of validity concerns issues regarding how the experiment performed is suited to answer the stated research questions.
- **Threats to internal validity.** This aspect of validity is usually the most notable one in empirical software engineering studies and deals with issues of correlation vs. causation. Specifically, these are issues regarding whether the observed effects are caused by the treatment or whether they are caused by other unidentified factors. The challenges of working with data from vulnerability databases described in the previous section largely fall within this aspect.
- **Threats to external validity – Generalization.** This aspect concerns issues regarding the generalizability of the results (e.g. to other software programs or other programming languages).
- **Threats to reliability.** This aspect refers to threats arising when the data collection or analysis are dependent on the specific researchers that performed the tasks. Such threats arise more often in human-subject studies (e.g. when the interpretation of answers may be subjective to the researchers).

In this dissertation, we address validity-related concerns in respective "threats to validity" sections for each of the three empirical studies we conduct.

3

AN EMPIRICAL STUDY ON THE MATURITY OF STABLE RELEASES

3.1 INTRODUCTION

This is the first chapter describing a novel contribution. In this chapter we investigate the first research question of Section 1.3): *Is the security of software improving over time?* We empirically investigate the hypothesis that there exist observable decreasing trends and patterns in the vulnerability discovery rate of stable releases of popular Free/Libre and Open Source Software (FLOSS) projects. Consistent with domain terminology, by *stable* we mean that only critical patches resolving discovered issues are applied during the lifetime of a release, with no new features introduced. In case we indeed observe such decreasing trends, the speed and characteristics of the maturing behavior (decrease in rate) can provide us indicators useful for assessing the development of software quality over time and even for making comparative arguments about different projects. Apart from 'engineering' metrics such as the ones mentioned above, 'economic' metrics revolving around monetary rewards for vulnerability discoveries have long been known to be a promising avenue towards quantifying the security quality of software [85, 99]. Therefore, in this chapter, we also seek trends in prices of bug bounties for FLOSS vulnerabilities that may corroborate the observed trends in the discovery rate and may also be used as indicators of quality.

This chapter extends on the content of a research article [3] published at the *ACM Transactions on Privacy and Security (TOPS)* journal. The scientific contributions presented in this chapter are:

- a dataset creation methodology that enables the study of vulnerabilities in stable releases of Debian GNU/Linux packages
- a large-scale (>10 000 CVEs) empirical study of vulnerabilities affecting stable releases of 1 195 FLOSS packages
- an empirical study on the development of rewards (bug bounties) awarded on a prominent bug bounty platform
- a critical interpretation of the presented results and a discussion of their implications

Chapter organization. This chapter is organized following the steps of the general methodology of Section 1.3. We first present the problem and state the specific research questions that we investigate in this chapter (Section 3.2). Then, we provide a compact overview of some background knowledge necessary for the comprehension of the content of the chapter (Section 3.3). Next, we provide an overview of the related work on the topic (Section 3.4) followed by a

detailed description of the dataset creation methodology (Section 3.5). We then present the results of our large-scale empirical study (Section 3.6) followed by a discussion on their implications (Section 3.7). Finally, we discuss some possible threats to the validity of our results (Section 3.8) before concluding the chapter (Section 3.9).

Availability

The code for the creation of the dataset, as well as for the analysis presented in this chapter is publicly available at `https://github.com/nikalexo/DVAF` under a free software license.

3.2 MOTIVATION AND RESEARCH QUESTIONS

There is a general feeling among security researchers and practitioners that the rise in the overall number of reported vulnerabilities in recent years can be primarily attributed to the sizeable increase in the number of software projects in use and their complexity [44], and hence the increase of the overall attack surface, while the quality of established and widely used software components may be improving. For instance, Ozment and Schlechter [84] reported a gradual decrease in the number of foundational (present at the initial release) vulnerabilities of the OpenBSD operating system. In this chapter, we set out to answer a fundamental question regarding the state of open source software security: *Is the security of software improving over time?* We approach this question from two angles, an 'engineering' and an 'economic' one[1].

* **An engineering angle.** We ask: Is software maturing over time? From this angle we primarily consider as *maturing behavior* a decrease in the vulnerability discovery rate for a given software program, although we also consider and discuss a few other indicators of quality (e.g. ratio of severe vulnerabilities). By *over time* we primarily consider whether a specific software release shows signs of maturity within a reasonable amount of time. We ask: *Are we finding and fixing vulnerabilities fast enough to have a measurable impact on security during the lifetime of a software release?* To supplement our insights we secondarily consider whether successive software releases show signs of maturity.

* Since software projects are generally evolving and their codebases are expanding, in order to get a best-case assessment of software maturity, we focus on vulnerabilities that affect *stable* releases of software packages. During the time of a stable release, only critical patches are applied (mostly security updates) and therefore our results will not be influenced by new vulnerabilities arising from the introduction of new functionality. Furthermore, focusing on stable releases also provides a best-case assessment of maturity when considering bias that may arise from an increase in vulnerability finding effort for a given software program. White-hat hackers (vulnerability researchers, testers,

1 Ozment [85] introduced the terms 'engineering' and 'economic' to describe notions similar to ours in his dissertation.

etc.) tend to focus and test newly released software versions and not versions that are included in stable releases (which, e.g. in Debian, have to go through a lengthy 'freeze' period). Therefore, only a subset of the vulnerabilities discovered will affect the stable releases of the software. Following this argument, we expect an increase in the vulnerability finding effort for a program to have less of an effect on its versions included in stable releases compared to newer versions.

* Considering all the above, we slightly modify and specify our 'engineering' question to now ask: Are stable software releases maturing over time? If we were not to discern signs of maturing behavior for stable releases, then we cannot hope to observe such behavior for constantly updated versions. On the other hand, if we were to observe such signs of maturity in the vulnerability reporting rate (e.g. a linear drop starting a year after release, or a drop in the reporting rate for new stable versions compared to previous ones), then we could use characteristics of the behavior (e.g. speed of decrease) as indicators of the security quality of projects. As an indirect consequence, such an observation would also support the social utility of vulnerability finding, in the sense that discovering and patching vulnerabilities would have a measurable impact on security.

  As a source of stable releases of software, we choose the Debian distribution of GNU/Linux. We choose Debian for our empirical measurement study due to three important characteristics: (a) it is one of the biggest (>50 000 software projects including most popular FLOSS, available as packages) and most popular collections of FLOSS in existence; (b) its policy of only adopting critical patches into the stable release makes it very amenable to testing our "maturing" hypotheses; (c) security is handled rather consistently, using a dedicated team with transparent workflows, public reports and status tracking. While our investigation is limited to Debian, it is likely that the results can be generalized to all general-purpose Linux distributions because the vast majority of software projects and code underlying the various distributions is identical: a vulnerability in a particular software suite will affect the respective package regardless of how it is distributed. In fact, our dataset likely underestimates the vulnerability rates of many other popular distributions since the Debian release policy is known to be rather conservative. However, the precise effects of program selection or release policy are not currently known and their investigation (e.g. via a comparative analysis of different distributions) is outside the scope of this dissertation. More information on the processes and release policies of Debian
* can be found in Section 3.3. Overall, from an 'engineering' angle, we ask: *Is software in Debian (Stable) maturing over time?*

* **An economic angle.** We ask: Are financial rewards for vulnerabilities increasing over time? The economic angle is based on the simple premise that in an ideal vulnerability market, as vulnerabilities would get more difficult to find, the reward for reporting them would increase. Although the potential of vulnerability markets to improve software security assessment has long been established (e.g. [85, 99]), few real-world schemes (so called *bug bounty platforms*) exist that would enable an empirical investigation. Since we want to relate results from

this angle to results from the 'engineering' angle, we focus on financial rewards for FLOSS vulnerabilities. As a source of such information we choose the publicly visible reports of the well-known Bug Bounty platform HackerOne[2].

HackerOne is a popular source of data for bug bounty research and has been used in important recent works that generally study the bug bounty ecosystem (e.g. [62, 119]). The Internet Bug Bounty (IBB) program[3] is a community-driven initiative to award rewards for bugs affecting important FLOSS[4] components, running on the HackerOne platform. The program started in late 2013 and is ongoing as of the time of writing, consisting of a number of projects targeting different software components. It is managed by an independent committee of security specialists and sponsored by technology companies and donations.

* Overall, from an 'economic' angle we ask: *Are financial rewards for FLOSS vulnerabilities in HackerOne increasing over time?*

* **Statement of hypotheses:** From the two main research questions formulated in the previous paragraphs we extract 3 hypotheses of increasing depth and detail, which we investigate in dedicated sections. Figure 3.1 provides a graphical summary of the process which led us to these hypotheses following an IBIS notation (see Section 3.3 for more information on IBIS maps). Confirming any or all of the hypotheses would provide evidence supporting the claim that the security of software is improving, on the one hand attesting to the increasing effectiveness of security processes, and on the other hand potentially providing metrics (e.g. speed of decrease) suitable for assessing the security of successive releases of the same software or even comparing different projects. Our three main hypotheses (the first two referring to the 'engineering' angle and the third to the 'economic' angle presented above) are:

  – *H1: Software in Debian is maturing over time when considering all vulnerabilities equally (Section 3.6.1).* In this hypothesis, by maturity we refer to a consistent (decreasing) trend in the vulnerability reporting rate during the lifetime of a Debian stable release (even also including the long-term support phase). As stated above, such decreasing behavior would suggest that it is getting more difficult to discover vulnerabilities affecting a specific version of the software, and thus the release is in some sense maturing.

  – *H2: Software in Debian is maturing over time when considering only severe vulnerabilities or vulnerabilities of certain types (Section 3.6.3).* With this hypothesis we want to investigate whether maturing behavior (in the same sense as in H1) is present when considering specific classes of vulnerability severity (e.g. "critical" vulnerabilities) or different vulnerability types (e.g. memory issues) and how it may differ from the general case. If there is significant maturing behavior for high severity vulnerabilities, then we could use this as an indication for quality. Furthermore, difference in the observed behavior be-

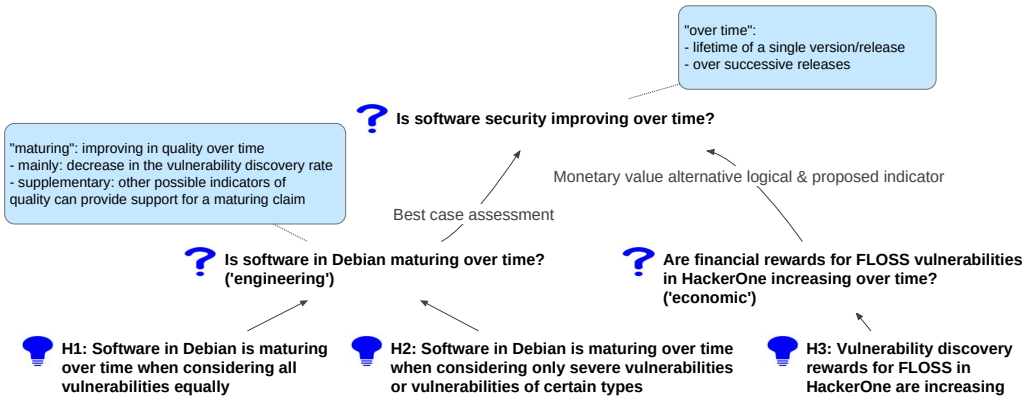
2 https://www.hackerone.com/

3 https://internetbugbounty.org/

4 with the notable exception of Adobe Flash until 2016.Perhaps surprisingly, Adobe's proprietary Flash Player was included in the program until August 2016 when it stopped with the argument that "Flash exploitation no longer has the same impact as when we started".

tween different types could identify vulnerability classes that may require additional attention.

– *H3: Vulnerability discovery rewards for FLOSS in HackerOne are increasing (Section 3.6.4).* Although not a traditionally widespread practice for FLOSS, there exist programs awarding bounties of the discovery of vulnerability in FLOSS projects. As with trends in the rate of vulnerability reports, (this time increasing) trends in the offered rewards offered for FLOSS vulnerabilities in the HackerOne bug bounty platform can provide indicators of increasing quality.



* **Figure 3.1:** Summary of how the hypotheses investigated in this chapter were derived from the higher level research questions. Produced following an adapted IBIS (Issue Based Information System) notation with the designVUE tool[5]. The main research problem is divided into two sub-problems and the hypotheses H1–H3 are depicted as positions/answers to these sub-problems.

3.3 SPECIALIZED BACKGROUND & TERMINOLOGY

In this section we briefly go over some necessary material for the comprehension of the contents of this chapter.

The Debian GNU/Linux distribution: Debian is a distribution of the GNU/Linux operating system including over 40 000 software packages, covering a significant portion of the most widely used FLOSS in existence [11] (for comparison 4 000 in Red Hat). All packages are open source and free to redistribute, usually under the terms of the GNU General Public License [104]. Debian officially supports 9 different architectures, and several other operating systems (e.g. Ubuntu, Raspbian) are based on it. It follows a conservative maturing release cycle aiming for maximum production-level stability and security for its `stable` release. A new stable version is released about every 2 years and only important patches are applied to the packages during its lifetime. In the meantime, developers and testers have time to examine and patch the newer versions of the packages to be introduced in the next stable release. In this phase, these packages comprise

5 https://www.imperial.ac.uk/design-engineering/research/engineering-design/engineering-knowledge-and-data/designvue/

the `testing` distribution. Debian releases are characterized by a number and a name, traditionally from Toy Story[6]. Security vulnerabilities are handled in a transparent manner by the Debian security team [28]. The security team reviews incident notifications for the stable release (only) and after working on the related patches, publishes a Debian Security Advisory (DSA). The DSAs contain detailed information on the vulnerability, including the affected packages and the corresponding CVE numbers.

Statistical tests: To support observations made via visual inspection of plots, it is often required to provide evidence that the observations carry statistical significance. Therefore, specifically in the field of reliability theory, the Laplace trend test has been traditionally employed to support evidence of a decrease in the rate of failures of a system. It tests whether the distribution follows a Poisson process or there is an increase or decrease in the rate of events (failures) taking place. Although the Laplace test is mathematically not entirely suitable for the scenario of vulnerability discoveries [85] (as it does not satisfy the independence requirement – see also Section 3.8), it has been widely used by seminal previous work [84, 94, 119] in the area and therefore it is also employed by us for reasons of comparability with previous work (always with a pinch of salt). In addition to the Laplace trend test, we fit linear models using ordinary least squares (OLS) to test for the significance of increasing/decreasing trends in histograms (e.g. CVEs per month). Although statistical tests are important to support our observations, they do not offer much to the presentation of the results, and as such, graphical representations of the Laplace tests, as well as the complete statistical test results are confined to Appendix B.

* **Issue based information systems (IBIS):** We already used an IBIS map to visually summarize the process of deriving the hypotheses investigated in this chapter (Fig. 3.1). To visually capture and summarize the results and argumentation presented in Section 3.6 (Results) we also use suitably adapted *issue-based information systems (IBIS)* [55] maps. IBIS maps are graphs that are often used to facilitate the understanding of complicated problems and to represent conversations as exchanges of arguments for/against a position. IBIS maps/graphs consist of four types of nodes: (i) questions/issues, (ii) answers/positions, (iii) arguments *for* a position, and (iv) arguments *against* a position. In our context these nodes correspond to: (i) higher level research questions, (ii) hypotheses, (iii) result/observation *supporting* a hypothesis, (iv) result/observation *contradicting* a hypothesis. A *primary* result/observation node can in turn contain other result/observation nodes which document further considerations for/against the hypothesis related to the argument of the primary node.

3.4 RELATED WORK

In this section we provide details on the prior work most related to our study. We also discuss how related work affected our research question selection and

6 For an overview of Debian Releases see `https://www.debian.org/releases/`.

study design choices (research questions and focus on Debian stable distribution as already introduced in Section 3.2).

* **Is security improving?** Our work is motivated by the results of Ozment and Schechter [84], presented more than a decade ago (2006). In the paper, the authors looked for evidence that the quality of software is increasing, by examining the vulnerability rate of the OpenBSD operating system. Their dataset consisted of 140 vulnerabilities, 87 of which were *foundational*, i.e. vulnerabilities that were part of the first release of the product. They concluded that the rate of foundational vulnerabilities is decreasing with time (statistically significant decrease after ~6 years), and presented this as an argument that the security of the software is increasing. This work came as a response to the 2004 paper by Rescorla [94], who reported no measurable evidence of an improvement in the security of software (w.r.t. the vulnerability discovery rate), and provocatively proposed that we should perhaps spend our time otherwise (in the sense that since vulnerabilities are seemingly infinite, finding and patching some of them does not really make a difference).

  Our study aims to investigate whether more than a decade later, Rescorla's concerns still apply (we discuss our insights regarding this in Sec. 3.6.1). It is interesting future work to see how the vulnerability rate and life span in the OpenBSD system has involved over time by recreating the measurements of Ozment and Schechter [84], however looking only into foundational vulnerabilities would provide limited insights since the amount of code of the first stable release of many FLOSS projects that is also part of the current stable release is
* minimal (e.g. for the Linux kernel). For this study instead, we choose to utilize a richer dataset (Debian) including most popular FLOSS components running on billions of devices. Furthermore, the duration of a stable release provides a natural time frame to seek for evidence of vulnerability depletion that would lead to a practically significant improvement in security. Therefore, we want to study the vulnerability discovery rate during a specific stable release (results in Section 3.6.2).

Vulnerability discovery models. Apart from the ones mentioned above, there are a lot of works on vulnerability discovery and lifecycle analysis, and the goal of this section is not to cover it all, but to go over those that connect with our work. Bugiel et al. [20] used weighted average models with constant weight, in order to predict vulnerabilities of Debian packages, with the goal to use the prediction as a trustworthiness score for the component. We follow a similar methodology to theirs in order to collect a part of our dataset, but our analysis differs in its goal (which is to investigate the hypothesis of maturing behavior). Clark et al. [23] studied the effect of code familiarity (i.e. a security tester being familiar with the code) in the vulnerability discovery process. They found out that there is a considerable period of time, called a "honeymoon", before the first vulnerability of a component is found, and then vulnerabilities are found at a much faster rate. Our research focus is complementary to theirs, as we want to study the subsequent stages of the vulnerability discovery process, from the moment a component becomes part of a stable distribution. Alhazmi and Malaiya [7] showed that time-based and effort-based vulnerability discov-

ery models (VDMs) are a good fit for the vulnerability time-series of Windows versions, indicating that these factors certainly affect the vulnerability discovery process. Contrary to their study, we want to investigate a much larger body of software over multiple stable versions. However, estimating the effect of effort in the discovery process could be a natural next step in the research line introduced in this chapter. Kim et al. [49] tried to create vulnerability discovery models of single versions of Apache and MySQL by taking into account the shared code of those versions with the following ones. They show that shared code between successive versions of a component may lead to an increase in the vulnerability rate of the earlier version, due to an increase in vulnerability finding effort (scrutiny) typical after the release of a new version. We want to investigate such effects further and select OpenJDK and PHP as two popular pieces of software to study that underwent major releases of new versions during the lifetime of a Debian stable release. Roumani et al. [95] fit several time-series models to the vulnerability rate of a number of software components, and reported reasonable prediction accuracy. Although we do not attempt any predictions, this could be an interesting future work direction.

Measurements in the wild. The line of work based on the WINE dataset [15, 74, 115] (a large dataset of Symantec anti-virus logs), which provides several measurements on attacks in the wild, is also relevant to our research. Specifically, Nappa et al. [74] studied the patching behaviours of users and focus on the issue of shared code between different software or different versions of the same software that can lead to successful attacks, even though users think they have patched a vulnerability. This work investigates a latter part of the vulnerability lifecycle (i.e. how fast users install patches after they have been made available), while we want to investigate an early part of the vulnerability lifecycle (i.e. the discovery rate and characteristics of new vulnerabilities). The insights provided by this work on the importance of shared code motivate us to investigate the issue of shared code in Section 3.6.1.

Other relevant studies. Li and Paxson [58] performed a large-scale study on security patches based on the NVD and commits in the projects' version control systems. They focused on static characteristics of security bugs (e.g. the amount of time they are present in the code, how this is affected by their type) and their patches, while we primarily want to investigate longitudinal effects, something not addressed by them. Thus, our study goals are complementary to theirs. We will discuss our findings regarding vulnerability types in relation to theirs in Section 3.6.3.

Shahzad et al. [103] performed a study focusing on exploitation trends. It is notable that for the overall vulnerability disclosure rate for the large set of software they considered (with disclosure dates in the range 1998-2011), they observed a declining trend after 2008. It is interesting to see whether we can also observe such a stable (even declining) trend for successive stable releases of Debian.

* Edwards and Chen [31] noted a decrease in the vulnerability rate of specific series of software releases after 3 to 5 years from the initial release. We want to investigate whether such a decrease is observable within the lifetime of a

stable release. Finally, Tan et al. [106] studied bugs in three open source projects, the Linux kernel, Mozilla and Apache. Their study looks at bugs of any kind, and security bugs in particular are discussed only briefly, mostly focusing on the effect bug types have on bug severity. We compare our results to theirs in Section 3.6.3.

* **Summary.** As evident from the above, there has been considerable prior work investigating the vulnerability discovery rate of software and trying to attain insights regarding, among others, how long it takes to observe maturing behavior (decrease in the vulnerability discovery rate) for software releases. Our study, as presented in this chapter, takes a new approach towards investigating such fundamental questions by focusing on vulnerabilities affecting packages included in stable (Debian GNU/Linux) distributions of software. As already stated in Section 3.2, investigating vulnerability rates through the lens of stable distributions will allow us to mitigate several sources of bias often affecting similar studies (see Section 3.8) and work towards a large-scale and practically relevant best-case assessment of software maturity.

3.5 DATASET CREATION METHODOLOGY

During the dataset creation process, an important goal was to construct a platform that would provide the means for researchers to validate and extend our results in a reproducible way. Therefore, we offer an extensible platform that automatically generates up-to-date datasets via parsing relevant repositories. We also package the relevant analysis scripts together with the dataset collection ones in a single repository. This Debian Vulnerability Analysis framework (DVAF), as we coined it, consists of two basic components: Data Collection and Data Analysis, as shown in Fig. 3.2. Our implementation consists of around 1 000 lines of python3 code and is available as open source on github[7]. We note that we used as a basis for our data collection scripts the Perl code that Bugiel et al. [20] made available to us. Below, we proceed to provide details about the dataset collection and cleaning process.

The currently implemented modules work as follows:

DSA collect: Debian Security Advisory text is automatically collected via downloading the html source from Debian's security information pages, and then applying the relevant filters and regular expressions to extract the names of the affected packages, the date of the advisory, and the associated CVE references. The URLs of DSA pages are of the form `https://www.debian.org/security/YYYY/dsa-NNNN`, with `YYYY` standing for the year when the DSA was reported, and `NNNN` for the unique DSA identifier.

DLA collect: Debian long-term support advisories are automatically collected by parsing the text of entries of the debian-lts-announce mailing list. The same information points as in the case of the DSAs are extracted from the mail text. DLAs are available over https with URLs of the form `https://lists.debian.org/debian-lts-announce/YYYY/MM/msgXXXXX.html`.

7 https://github.com/nikalexo/DVAF

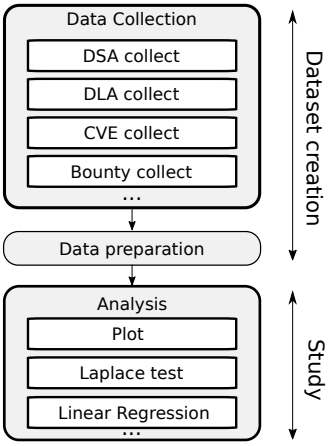


Figure 3.2: DVAF's extendable architecture and workflow

CVE collect: CVE data, including the date of the CVE, and various metadata (CVSS score, CWE type etc.) are collected by employing the `cve-search`[8] tool, a tool for local lookups on reported CVEs.
Bounty collect: Bug bounty data are collected by scraping the HackerOne platform's publicly visible portion. All available information is obtained (product affected, date, bounty amount, etc.) by utilizing the platform's API[9].
Data cleaning and preparation: Note that by mapping vulnerabilities from the DSAs/DLAs to the CVEs (i.e. relying on DSAs and DLAs for information regarding which CVEs affected which package versions), we already avoid common errors in the NVD, such as versioning issues. The preparation of the data consists of manual corrections of known mistakes in the vulnerability reports, further dealing with package versioning and possible date differences between the DSA and CVE reports, and formatting the data in a standard, transferable format.

For example, DSA-2103 does not include a CVE reference, although CVE-2010-3076 references the DSA and matches its description. We identified a number of such issues. Furthermore, due to trademark issues regarding the Mozilla logo, Mozilla products were distributed under alternative names in Debian from 2006 to 2016 (Iceweasel instead of Firefox, Iceape instead of Seamonkey, Icedove instead of Thunderbird). Also, some packages have names based on the software version they distribute (e.g. php5 and php7.0), while others, such as the Linux kernel have changed their naming conventions over time (from kernel-* to linux).

Note that the manual effort is a one-time process, as appendable lists with package name changes etc. are maintained as configuration files, and the rest of the process is automated. As the date of vulnerability disclosure, we choose the earliest of the dates reported in the DSA and the corresponding CVE. Vulnerabilities for source packages are grouped by month and a time-series of vulnera-

8 https://github.com/cve-search/cve-search
9 During the revision process of the paper [3] where the contributions of this chapter were first presented, we noticed that the new HackerOne API is now only available for a fee, whereas an older version of the API was available for free during our study.

bility incidents is created for each source package. For most of our studies, we created a single time-series for all the versions of a package using regular expression rules (e.g. in the overall analysis all vulnerabilities that affected Debian packages php5 and php7.X appear under the package name of php7.0 – always counting each CVE once), however this can be configured. All data points are stored in JSON text representations of python dictionaries. In total the dataset we used for the analysis in this chapter contains 10 716 CVEs spanning from 2001 to 2018.

Analysis functionality: To check our hypotheses, we developed a number of analysis and plotting scripts and made an effort to render them re-usable to the extent possible. The basis of the framework can be used for other studies or as a starting point for software security metrics and risk assessment methodologies.

3.6 RESULTS

In this section we present the results of our large scale empirical measurement study.

3.6.1 *Data overview and distribution*

We start off with an overview of the Debian ecosystem w.r.t. its security characteristics before we proceed to investigate our hypotheses. These more general observations set the context of the study and will help us with the interpretation of our results in later sections. Note again that contrary to previous studies (e.g. [7, 23]), we do not investigate the vulnerability rate of specific versions of software during their development cycle and immediately after their introduction, but the software versions that are included in the corresponding stable releases of Debian.

Global observations: Table 3.1 presents the 20 top vulnerable Debian source packages between 2001 and 2018. An automated procedure was established to collect the vulnerabilities reported for previous versions of a package and attribute them to its current version in the stable distribution. Manual checks and small corrections were subsequently performed (again included in the framework as ready-to-use configuration files – see Section 3.5). The Linux kernel is the most vulnerable component, followed by the two main browsers in use (Chromium[10] and Firefox). The total number of unique vulnerabilities[11] reported in the 18 year period was 10 716, with the kernel accounting for around 9 % of the total (948). During the years 2017-2018, a total of 2 366 vulnerabilities were reported, with Chromium being by far the most affected package, accounting for 303 vulnerabilities (around 13 % of the total) compared to the next most vulnerable package (the Linux kernel) which was affected by 160 (around 7 % of the total).

10 open source code-base of the proprietary Google Chrome browser – among others.

11 Note that a vulnerability may affect more than one packages. More discussion on this follows in this section. Also note that we count vulnerabilities by their CVE identifiers (1 CVE id = 1 vulnerability).

	2001–2018		2017–2018	
package	number CVEs	rank	number CVEs	rank
linux	948	1	160	2
chromium	799	2	303	1
firefox-esr	739	3	147	3
icedove	600	4	127	5
php7.0	386	5	28	19
openjdk-8	309	6	89	7
wireshark	303	7	43	14
xulrunner	211	8	–	–
mysql	209	9	47	12
imagemagick	195	10	99	6
iceape	178	11	–	–
xen	172	12	59	8
tcpdump	156	13	131	4
wordpress	147	14	46	13
openssl	134	15	13	29
tiff	127	16	55	10
qemu	121	17	36	16
mariadb	116	18	51	11
ruby2.3	84	19	39	15
graphicsmagick	82	20	56	9

Table 3.1: The top twenty packages with the most vulnerabilities (counted by CVEs) in time periods (i) 2001-2018 and (ii) 2017-2018.

In Fig. 3.3, we see the distribution of vulnerability reports (CVEs) among source packages of Debian for the years 2001-2018[12]. In the figure, packages that had at least two vulnerabilities in the specified time frame are included, yielding a total of 634 source packages. An additional 561 source packages had a single vulnerability and were not included in the figure for readability reasons (the complete figure is available in Appendix B).

The rich club effect: Interestingly, the distribution is characteristically heavy-tailed (notice that the y axis is logarithmic) with a few packages dominating the total vulnerabilities reported and a long tail of a large number of packages with only a few vulnerabilities. Inspecting the plot (Fig. 3.4) of the probability distribution of the data in (double) logarithmic axes, we can observe a near-straight line, indicative of a potential power-law (Zipfian) distribution. Power-

12 Our analysis in this chapter is limited to data until the end of 2018 since this was the last completed year at the time we performed the analysis.

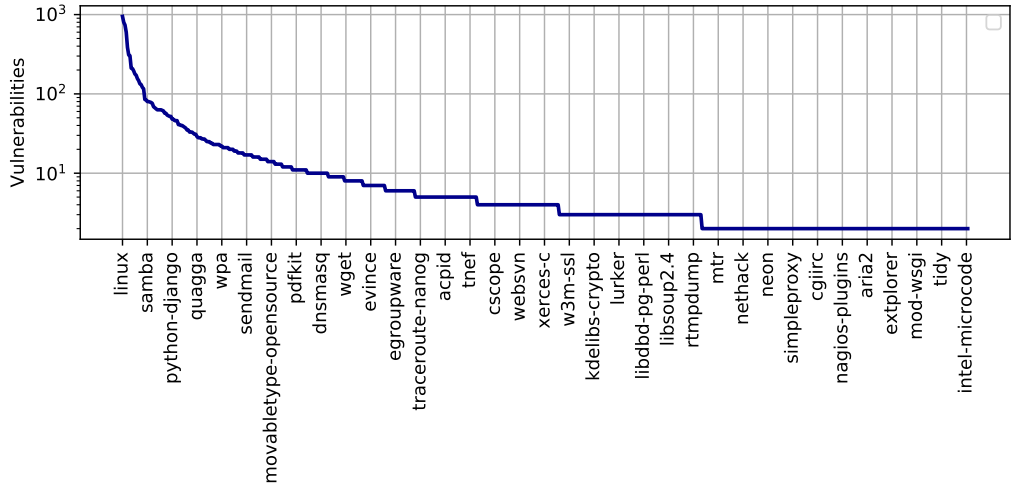


Figure 3.3: The distribution of vulnerabilities per package (years 2001-2018). Every twentieth package name appears on the x axis for space reasons. The y axis is logarithmic. Packages with at least two vulnerabilities are taken into account.

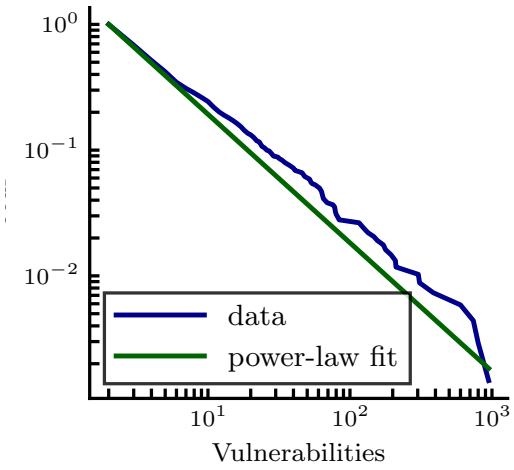


Figure 3.4: A log-log plot (complementary cumulative distribution function) of the distribution of Fig. 3.3.

laws are heavy-tailed distributions that are the result of generative mechanisms like scale-free networks or the distribution of wealth in society (Pareto distribution). Following detailed statistical testing using the seminal methodology of Clauset et al. [24] and the `powerlaw` statistical package [10], we fit a power-law of the form $p(x) \sim x^{-k}, x > x_{min}$ with $k = 2.02$ and $x_{min} = 2$, where x is the number of vulnerabilities of a given package, and p the probability density function[13]. In short, we observe that the majority of vulnerabilities is discovered in a small set of packages, with the rest contributing little to the total number. Although the distribution of vulnerabilities fits a power-law, other heavy-tailed

13 It is common in literature (e.g. [113]) to ignore the light lower tail and focus on the heavy upper tail when investigating potential heavy-tailed behavior. In our case, the best fit was achieved for $x_{min} = 9$ with $k = 1.94$, but the fit for $x_{min} = 2$ was deemed good enough and explained most of our data points.

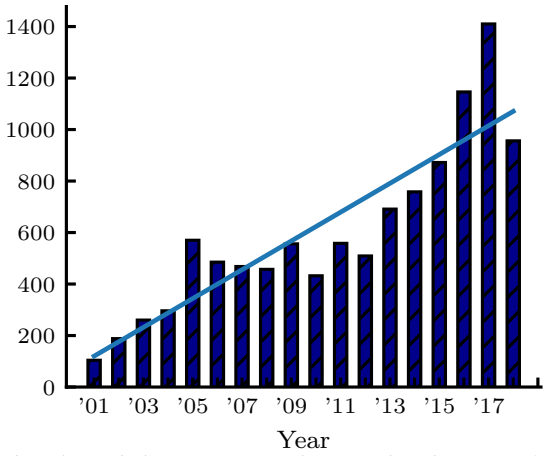
distributions (power-law with exponential cut-off, lognormal) are also possible fits, and in general very difficult to statistically disprove [24].

One should not jump to conclusions regarding the mechanism(s) that generate this distribution. Most likely, a combination of mechanisms leads to our observations, including economic, social and time-dependent factors. However, the hypothesis that the vulnerability distribution is attributed entirely to the size (in KSLoCs) of the packages was indeed statistically disproved (size does not follow a heavy-tailed distribution; Fig. B.3 of the appendix shows an intuition of this). A high-level generative mechanism of preferential attachment (the rich get richer), supporting a classic power-law, could be based on "the more we look the more we find" argument, where more bugs being found for a component cause more focus on the component, and thus in turn yet more bugs are found. Another fitting distribution is the power-law with exponential cut-off, where the rich get richer up to a point. This would explain the case where there is a limit on the rate of bug-finding for each package even if more resources are dedicated to the task. Both of the aforementioned related heavy-tailed distributions are possible fits with similar generative mechanisms and collecting more data as time progresses is required for more definitive statistical tests.

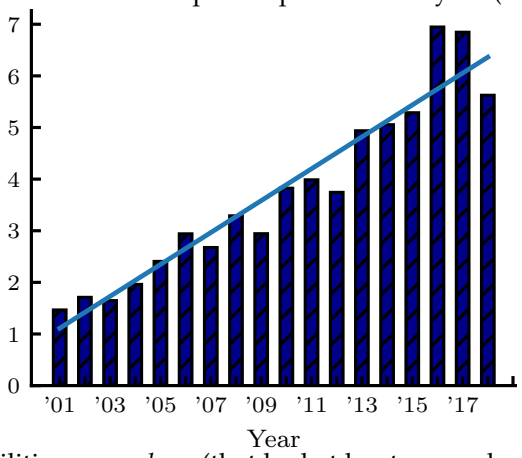
3.6.2 *Vulnerability trends in Debian (H1)*

In this section we investigate Hypothesis 1: *Software in Debian is maturing over time when considering all vulnerabilities equally*. We want to see whether there are signs that the quality of software (w.r.t. security bugs) is increasing over time; in other words, whether we have reached the point where the vulnerability discovery rate in stable versions of popular software is decreasing. In this section we do not discriminate based on a vulnerability's severity or type.

Concerning the total number of vulnerabilities reported in the Debian ecosystem w.r.t. time, Fig. 3.5a shows a clear upward trend as the years go by. Can this mean that the quality of the software (w.r.t. security) is decreasing, despite the considerable effort of developers, security researchers and professionals? One could argue that the amount of software packages in Debian increased dramatically in recent years and that this is the cause of the increase in the total amount of vulnerabilities reported. Thus, even one or two bugs that slipped the security measures of the individual development/testing teams, would contribute to a big yearly sum. That would be a reasonable explanation, as the stable version of Debian released in 2002 (Woody) contained only 8 500 binary packages, going up to 18 000 packages with the release of Etch in 2007, significantly increasing to 36 000 in 2013 (Wheezy) and peaking at 52 000 with Stretch, which was released in June 2017, and at 58 000 with the latest (at the time of the investigation) stable release named Buster, which was released in June 2019. However, we found evidence supporting the opposite. Interestingly, the number of vulnerabilities per package (among the packages that had a vulnerability reported for the specified year) also follows an upward trend, a fact obvious in Fig. 3.5b. In this figure we can even see a smoother, clearer upward trend compared to the previously presented Fig. 3.5a, although the slope of the trend is nearly identical.



(a) Total vulnerabilities reported per calendar year (2001-2018).



(b) Average vulnerabilities *per package* (that had at least one vulnerability report in that year) per calendar year (2001-2018).

Figure 3.5: Vulnerabilities: distribution and trends.

These observations, together with our previous assessment that the distribution of vulnerabilities among the packages can be attributed to a power-law generation mechanism, indicate that there are specific packages that continue to have large numbers of vulnerabilities for prolonged periods of time. As we can see in Table 3.1, the total is dominated by major projects, some of which we like to think of as leaders in the practice of secure software development. What is the explanation for this phenomenon?

Do "stable" releases mature?: A typical explanation would be that vulnerabilities were induced by software upgrades and the number of vulnerabilities affecting a specific version of a package gradually dropped to zero. An intuitive hypothesis would be that at least for certain stable versions of a package, the rate of vulnerabilities will eventually decrease. In order to test the claim that specific versions of a package reach a relatively secure state (few vulnerabilities reported per quarter) and that subsequent vulnerabilities that are attributed to the package are caused by updates, we perform a case study on two popular packages, namely PHP and OpenJDK. We pick these two packages because (a) they are widely used, and (b) they recently underwent major version changes

(translated to significant differences in their codebase, in contrast to other packages like the Linux kernel which follow a very smooth version transition). The hypothesis is that each major version of a package becomes more secure as time passes, as a result of the hard work of the security community and that a considerable amount of new vulnerabilities affect only the new code inserted with the major updates. To test this hypothesis we inspected the vulnerabilities reported for the newer versions of the packages and checked if they also affect older versions.

(Case study on PHP)
PHP is a popular server-side scripting language that is used by 79% of all websites whose server-side programming language is known[14]. We will look into the transition from php5 to the next version php7 [15] (php6 never made it to the public). The vulnerability history of php5 (see Fig. 3.6) indicates that the component is relatively hardened at the time the next version is released. The vulnerability discovery rate is relatively low and stable for the last months before the launch of php7. To support our hypothesis, we would expect a good

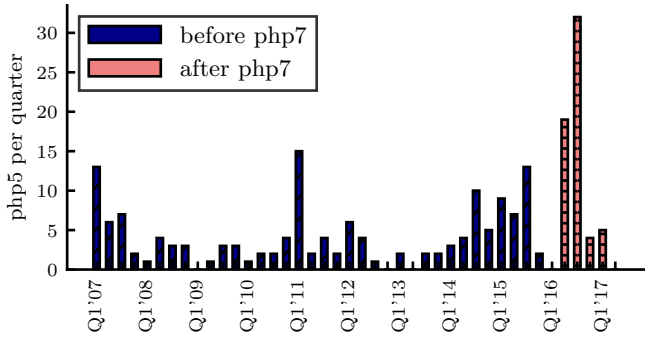


Figure 3.6: Vulnerabilities of php5, during its presence in stable releases, before and after the introduction of the next version (php7) in testing. Vulnerability rate: (a) before the launch of the new version: ≈ 4 vuln./quarter; (b) after the launch of the new version: ≈ 10 vuln./quarter.

amount of vulnerabilities after this point to affect the new version (php7) of the software but not older versions (php5.x). However, while we can indeed observe a substantial spike of vulnerabilities, most of those also affected the previous version (php5.x). The launch of the new version may have instigated researchers and bug hunters to look for vulnerabilities induced by the new code, but instead what they found were already existing vulnerabilities from previous versions - so called *regressive* vulnerabilities. After detailed inspection of all security incidents tracked by the Debian Security Bug Tracker[16], we found that in the time window of January 2016 - January 2018, out of the 103 vulnerabilities that affected php7, 81 (79%) also affected version 5 of the software[17].

14 https://w3techs.com/technologies/details/pl-php (July 2020)
15 Official package name php7.0
16 Relying on affected versions information available at `https://security-tracker.debian.org/tracker/`
17 Attribution of vulnerabilities to affected versions was made according to information about patched versions in the Debian Security Bug Tracker.

Further investigation regarding the nature of these vulnerabilities shows that a great portion of them are usual programming mistakes (e.g. input validation errors or integer overflows). Hence, their discovery may not be associated with any advances in the security tools used, rather it may reasonably be attributed to the fresh eyes that reviewed the code of the newer version (increase in interest – expended effort in bug finding).

(Case study on OpenJDK)
OpenJDK is an open source implementation of the Java Platform (Standard Edition), and since version 7, the official reference implementation of Java. Version 7 was introduced into the testing distribution of Debian in September 2011 and became part of stable in May 2013 (Debian Wheezy). It remained part of the stable until the release of Stretch (June 2017). The next version, OpenJDK-8, became part of the testing distribution in May 2015 and became part of stable with Debian Stretch (June 2017), replacing version 7. In Fig. 3.7, we see the vulnera-

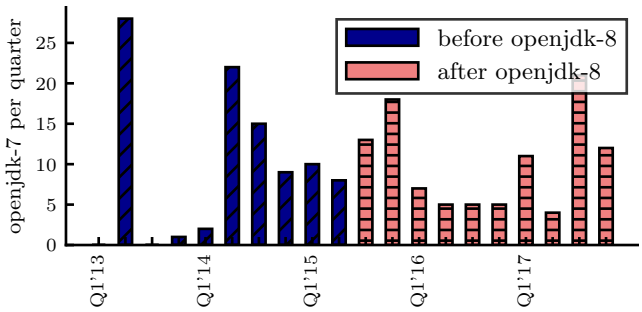


Figure 3.7: Vulnerabilities of openjdk-7, during its presence in the stable release, before and after the introduction of the next version (openjdk-8) in testing. Vulnerability rate: (a) before the launch of the new version: ≈ 11.3 vuln./quarter; (b) after the launch of the new version: ≈ 10.6 vuln./quarter.

bilities of version 7 before and after the introduction of the next version. Again, there is no statistically significant decline in the rate of vulnerability reports, and the introduction of the next version seems to contribute to the discovery of vulnerabilities of the previous version. To put things into perspective, out of a total of 38 vulnerabilities that were reported for openjdk-8 in the time span of June-November 2017, only 2 did not affect version 7, and most of them (31/38) also affected version 6, released almost 10 years prior.

(Case study on Debian Wheezy)
Although the detailed investigation of vulnerabilities for PHP and OpenJDK gave us some useful insights about the current state of software quality, these results cannot be generalized to other packages (also due to different release policies). In order to get a more complete view of the effect of new vulnerabilities on older versions, we study the security history of Debian 7 (Wheezy) that was released in May 2013 and was supported until recently by the LTS[18] team

18 Starting from 2014, Debian includes a Long Term Support (LTS) program, in order to extend support for any release to at least 5 years in total. For this investigation we also utilize Debian long-term support vulnerability advisories (DLAs).

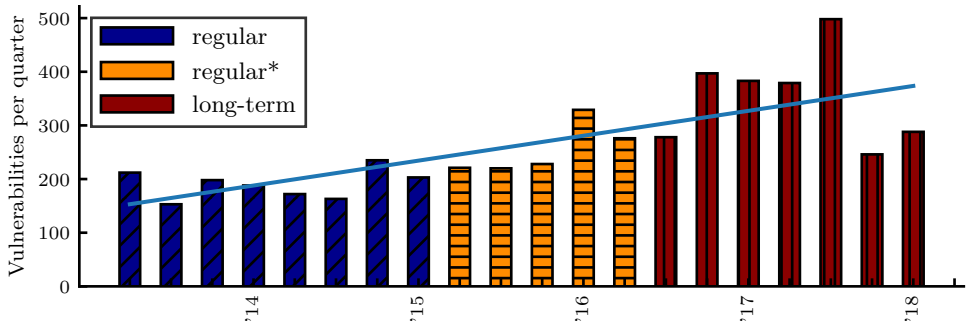


Figure 3.8: Vulnerabilities that affected packages of the Wheezy Debian release.
From Q2/2015 to Q2/2016, both Debian 7 (Wheezy) and 8 (Jessie) were supported by the regular security team. This was due to the fact that current Debian practice is that when a new stable version is released, the previous one (now codenamed oldstable) is still supported by the regular security team for another year and then passed to the LTS team. Therefore, the amounts of the regular period are a higher bound, as some vulnerabilities may have affected only the newer release. We note that in the LTS phase, only one release is supported at a time.

(May 2018). In Fig. 3.8, we see the distribution of vulnerabilities per quarter, starting from the release of Wheezy.

Even for a specific stable release of Debian, we can observe a clear upward trend that continues in the LTS phase of the software. These results support our findings for individual packages and show that the rate of vulnerabilities is not decreasing, and to the contrary is slightly increasing over time, even for a specific stable release over its whole lifetime of 5 years. This is quite an unexpected result, as we see that there are no signs of depletion of the
* vulnerability pool of a distribution as a whole even after 5 years. More research may be beneficial in order to individually analyze a larger number of packages and potentially identify differences, e.g. between smaller and larger packages or more popular and less popular packages. More detailed per project analysis is beyond the goals of this chapter and therefore we leave such analysis for future work.

The shared code effect: Shared code between applications (packages) can lead to the same vulnerability affecting more than one of them. Since shared code has been shown to be an important attack vector [74], we move on to investigate the prevalence of shared vulnerabilities in Debian. Of a total of 10 716 vulnerabilities affecting Debian packages, 2 462 affect more than one package. In Fig. 3.9, we see the number of vulnerabilities that affected at least two packages over time, and in Table 3.2, the most prevalent package sets jointly affected by vulnerabilities.

We observe that the package sets of jointly affected packages are dominated by Mozilla products and the duo of mariadb and mysql. This is attributed to the well-known fact that Mozilla products share a large portion of their source code (referred to as *Core modules*[19]), and mariadb starting as a fork of mysql. Although for all results presented in this chapter we count each vulnerability only once even though it may affect one or more packages, we did not see any

19 https://wiki.mozilla.org/Core

package sets	vuln. #
firefox, icedove	363
firefox, icedove, iceape	87
mariadb, mysql	85
firefox, icedove, iceape, xulrunner	57
firefox, iceape, xulrunner	28
firefox, icedove, graphite2	23
icedove, xulrunner	16
xpdf, kdegraphics	15
php7, file	13
imagemagick, graphicsmagick	11

Table 3.2: Most common sets of packages jointly affected by vulnerabilities.

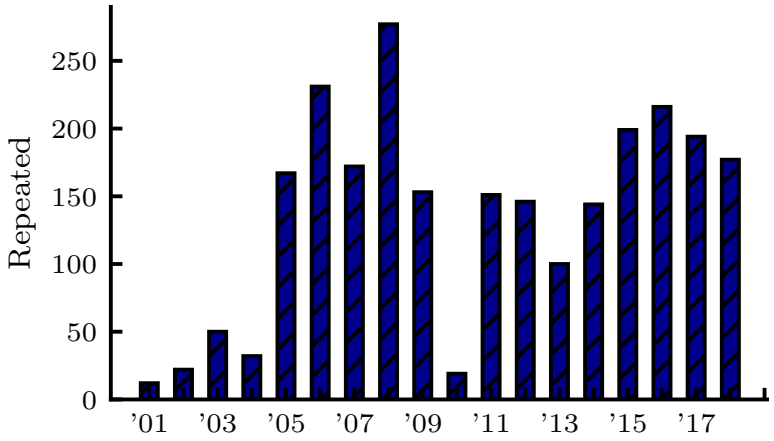


Figure 3.9: Vulnerabilities affecting at least two Debian packages.

qualitative differences when counting vulnerabilities multiple times in any of the trends we investigated. Therefore, we can conclude that the shared code effect does not significantly affect the overall trends of vulnerability discoveries in Debian.

Discussion on the results: In this section, we investigated the longitudinal development of vulnerabilities in the Debian ecosystem. After an examination of the vulnerabilities reported for the distribution overall (Fig. 3.5), individual widely used packages (case studies on PHP and OpenJDK – Figs. 3.6-3.7), and a specific stable release of Debian (case study on Debian 7 – Fig. 3.8), we did not observe signs that the vulnerability discovery rate decreases over time. To the contrary, we discerned a generally increasing rate of vulnerabilities overall. Therefore, our observations contradict our hypothesis of maturing behavior.

One interpretation could be that we are still in the phase of *the more we look - the more we find*. Although automated security tools and manual security in-

spection are becoming more widespread and effective, we have not reached the point of curbing the vulnerability rate yet.

Our results allow us to draw interesting comparisons to studies performed over a decade ago. Rescorla claimed [94] that there was no clear evidence that finding vulnerabilities made software more secure, and that even the opposite may be true, i.e. that finding vulnerabilities, given that their rate is not decreasing, leads to more risk than good, by allowing hackers to attack unpatched systems. Another study from 2006 by Ozment and Schechter [84] tried to challenge Rescorla's claims and found evidence of a decrease in the vulnerability rate of the foundational (in the code since the first version) vulnerabilities of OpenBSD in a 7.5 year interval (statistically significant decrease was observed after 5 years from the release of the sotware). As stated above, in our study we did not observe such decreasing trends. Of course, we did not focus on foundational vulnerabilities since most of the examined software projects are rapidly evolving and foundational vulnerabilities would provide no actionable insights. Our results show, that more than a decade later, the security of Debian as a whole, and for PHP and OpenJDK individually, is not increasing. After the impressive growth of the security community since 2006, we still either have not reached the point where vulnerabilities have become substantially more difficult to find, i.e. we have not seen signs of software maturity. Here, we have to note that an alternative optimistic interpretation where the effort expended in the vulnerability discovery process constantly increases over time, would also fit our results, even if the quality of the software is in fact increasing. In that case, we would expect to observe maturing behavior at some point in the future, assuming that vulnerabilities are finite. Therefore, recreating the results in this section in the future would be worthwhile.

Our results can also be compared to the more recent ones of Edwards and Chen [31] from 2012. By investigating the vulnerabilities of Sendmail, Postfix, Apache httpd and OpenSSL, they conclude that although the quality of the software under question did not always improve with each new release, the rate of CVE entries generally begins to drop three to five years after release, indicating a stage of maturity of the software. Our results do not disprove the fact that the vulnerability rate of a specific version may begin to drop three to five years after its release. However, for the software packages of our case study (PHP and OpenJDK), it increased again when the new version entered the testing phase, likely due to a renowned interest in the new version as a testing target combined with a significant amount of shared code between the versions. This hints to the fact that testing scrutiny is a dominating factor of the vulnerability discovery process, as "fresh" eyes looking at the code seem to be able to find additional vulnerabilities. This comes back to our *the more we look the more we find* interpretation of the results presented in this section. It is worth noting that this interpretation also implies that *the less we look the less we find*, meaning that we expect software programs that are not notable targets of testing efforts to have very few vulnerabilities discovered.

In another work, Clark et al. [23] found that, out of all the primal vulnerabilities (i.e. first exploitable vulnerabilities to be reported for a software release)

discovered, 77% of them affected earlier versions of the software. This result, along with our observations, indicates that the difficulty of finding a specific vulnerability may be subjective, and may vary considerably among different researchers/testers. This would be an explanation of why focus on a new version of a package leads to the prompt discovery of a majority of vulnerabilities also affecting the previous version (that had not been discovered before). Conclusions made in [62] that the unique characteristic of each individual offers unique bug-discovering potential and that each individual can expect to find a bounded number of bugs, further support this claim. In short, the process is still more of an art than a well-defined procedure, and the impact of automated tools requires further investigation.

Looking into the bigger picture of software engineering, we can find interesting relationships between our results and the Laws of Software Evolution, proposed by Lehman in the 1970's [56] and revised in the 1990's [57], with the addition of, among other, the Law of Declining Quality. This law states that "The quality of E-type systems[20] will appear to be declining unless they are rigorously maintained and adapted to operational environment changes". Our observations could be interpreted as showing that we have not yet achieved an adequate degree of rigorousness in our development and security processes, since the vulnerability rate of stable software versions does not show signs of decrease.

* **Summary:** In Fig. 3.10 we provide a graphical summary of the main results of this section in the form of an IBIS map. There are three primary results/arguments against the investigated hypothesis. Each of those arguments has a number of further negative (against the hypothesis) or positive arguments (in favor of the hypothesis) attached. Overall, we found no convincing evidence to support H1.

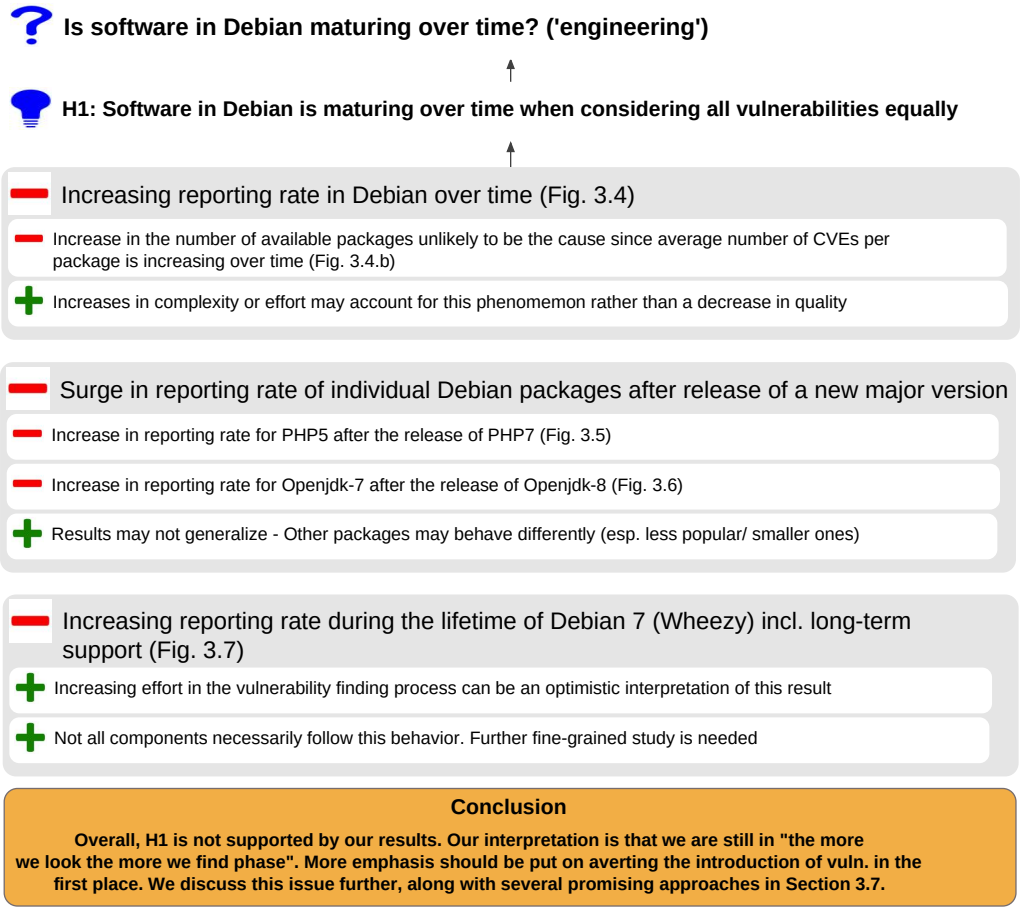
3.6.3 *Vulnerability Severity and Types (H2)*

* In this section we move on to investigate the same hypothesis as in the previous section (that software in Debian is maturing over time) taking into account vulnerability severity (H2.1) and type (H2.2).

* **H2.1 Software in Debian is maturing over time when considering only severe vulnerabilities:** Since we established that there is no observable decrease in the overall vulnerability rate of FLOSS, we proceed to investigate the hypothesis that although more bugs are discovered, they are less critical and more difficult to exploit than before. We formulate the generally stated hypothesis above into two specific hypotheses: (a) *the ratio of high-severity vulnerabilities decreases over time compared to less critical ones*, and (b) *the vulnerability discovery rate for critical vulnerabilities decreases over time*.

In Fig. 3.11, we see the progression of the ratio of low, medium, and high severity vulnerabilities, as classified by their CVSS score. An obvious trend

20 E-type systems are, according to Lehman, real-world systems influenced by the environment and people.



* **Figure 3.10:** Summary of main results of Section 3.6.2 in the form of arguments pro/against the investigated hypothesis. Produced following a relaxed IBIS (Issue Based Information System) notation with the designVUE tool.

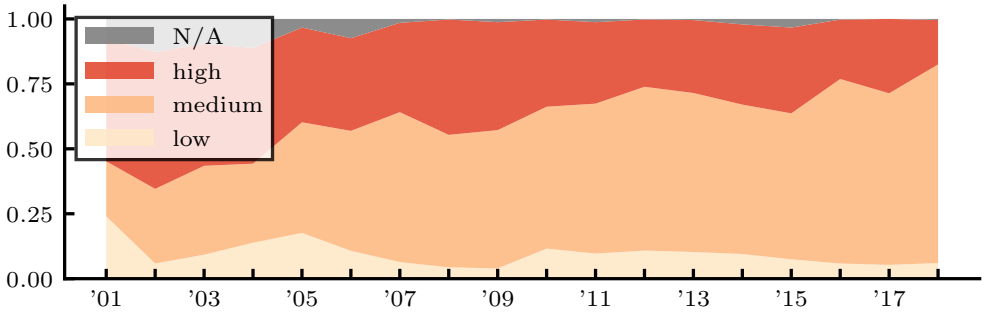


Figure 3.11: Vulnerabilities severity of the stable release over time.

of domination of medium severity vulnerabilities is observed, with a gradual decrease of the percentage of high and low severity vulnerabilities. We can also see that low severity bugs represent a very small percentage (under 10%). This can be attributed to the fact that the Debian security team only issues advisories for bugs that warrant immediate patching, and often low severity ones are left to be fixed as part of the normal release cycle of the package. We saw that

the percentage of high vulnerabilities shows a decrease recently compared to normal and low severity ones, but are high severity vulnerabilities becoming rarer?

To test this hypothesis, Fig. 3.12 shows the high-severity vulnerabilities discovered during the whole lifetime of Debian Wheezy, including its LTS period. It is evident that no decrease is observable, and to the contrary a statistically

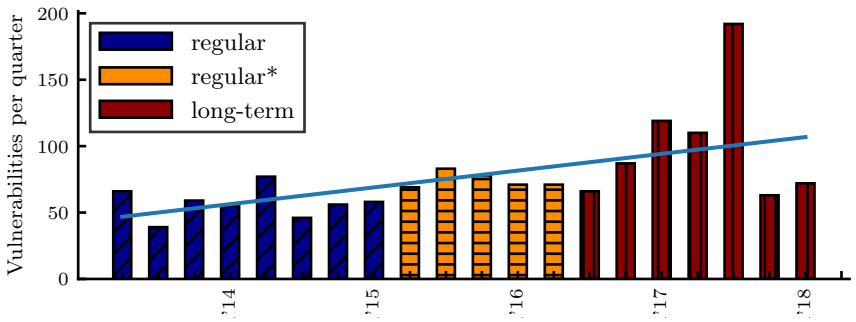


Figure 3.12: High severity vulnerabilities of Debian Wheezy. The irregular peak of Q3'17 can be largely attributed to DLA 1097-1 which contained 86 CVEs affecting tcpdump. During the *regular** period 2 releases were concurrently supported by the security team.

significant increase in the vulnerability discovery rate is observed until the end of the release's lifetime. Therefore, there is no sign of maturity even when only considering critical vulnerabilities of a stable release of Debian. This gives an overall picture, however it does not mean that all components necessarily follow this trend. More fine-grained study and comparison of the behaviour of the different packages may offer interesting results, however, as already stated such comparisons fall outside the goals of this chapter and are left for future work.

H2.2 Software in Debian is maturing over time when considering vulnerabilities of certain types: If software development is just too fast and tools are still limited and not widely applied, are we at least making progress on some part of the problem? Are certain vulnerability types being eliminated as a result of better practices and tools (e.g. more secure web programming, fuzzing tools)? To test this hypothesis, we investigate the distribution of vulnerabilities over time according to their types. Vulnerability type information is derived from the "Research Concepts" view (CWE-1000) of the Common Weakness Enumeration (CWE) list. According to the CWE documentation, this view is mainly organized according to abstractions of software behaviors and is intended to facilitate academic research into weaknesses. It follows a deep hierarchical organization where all vulnerabilities can be traced back to 11 root classes. In our study we matched each vulnerability that was attributed a CWE number with its root class(es). The 7 classes with a significant number of bugs are presented in Table 3.3.

The progression of bug types over time is shown in Fig. 3.13. The plot starts from year 2008, as this is the time where type classification of bugs started becoming standard practice of the NVD. Two observations are made. First, a

root CWE	Description
682	Incorrect Calculation, e.g. Integer Overflow
118	Incorrect Access of Indexable Resource ('Range Error'), mostly Buffer problems
664	Improper Control of a Resource Through its Lifetime, e.g. Information Exposure, Improper Access Control
691	Insufficient Control Flow Management, mostly Code Injection, Race Condition
693	Protection Mechanism Failure, mostly Improper Input Validation (CWE-20)
707	Improper Enforcement of Message or Data structure, mostly Improper Neutralization (SQL injection, XSS)
710	Improper Adherence to Coding Standards, mostly NULL Pointer Dereference

Table 3.3: Vulnerability type classification per root CWE number with most dominant examples in our dataset.

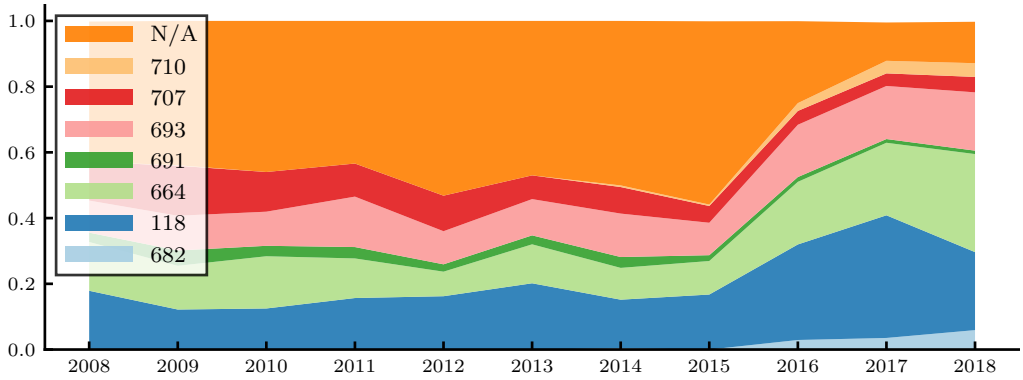


Figure 3.13: Vulnerability types per year of Debian stable. Labels correspond to root CWE numbers (research view).

big portion of bugs (N/A) did not fall directly under a CWE-1000 root class, especially before 2015. This is because many vulnerabilities (especially older ones) were classified in broad categories, such as CWE-16 "Code weakness" and CWE-17 "Configuration weakness", which are not compatible with the classes of the Research Concepts view, and according to CWE suggestions should not be used for mapping bugs – however NVD still maps to them anyway. Positively, in recent years the portion of unmapped bugs has fallen to under 20 % of the total. The second observation is that three types capture most of the classified bugs, namely Memory Index (118), Improper Resource Control (664) and Protection Mechanism Failure (693) errors account for more than 70 % of all Debian bugs in 2017, with their ratio relatively stable since 2008. On the other hand, message structure enforcement errors (707), in most cases improper neu-

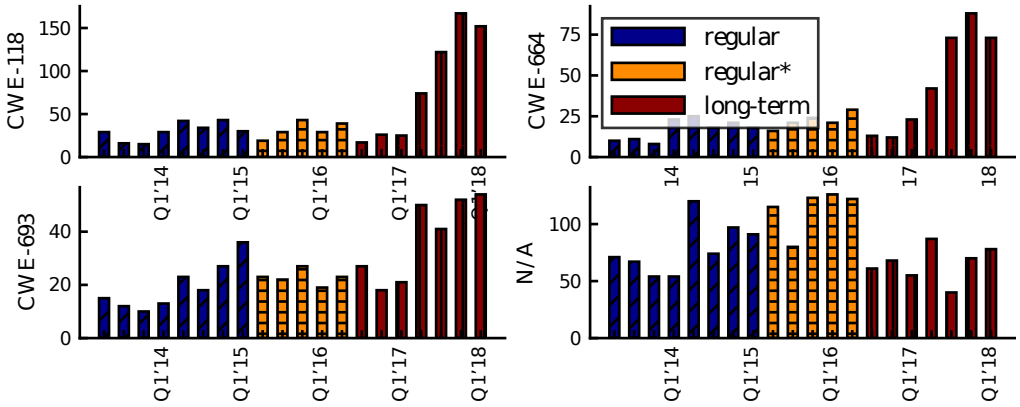


Figure 3.14: Main vulnerability types of Debian Wheezy, including LTS.

tralization of special characters leading to SQL injection (SQLI) or Cross Site Scripting (XSS), show a decrease in their prevalence from more than 10 % in 2008-2010 to a negligible portion in 2016-2018. This may be a sign that at last some maturity has been achieved for this specific bug type, which is reasonable as these vulnerabilities are suitable for automatic detection. In fact, this result supports the claims of [14], that automated black-box tools were effective at finding XSS and SQLI since 2010.

We move on to test for trends in the absolute number of vulnerabilities per type, rather than their ratio. In Fig. 3.14, the absolute number of bugs of the three most prevalent types plus the unclassified ones, are shown for the whole lifetime of the Wheezy release. No decreasing behavior can be observed for any of the four types, and especially memory errors seem to increase dramatically during the LTS period of Wheezy. However, no strong claims can be safely made regarding this, as the incompatibility issues of the CWE directives and the NVD classification hinders reasoning. On the positive side, the number of unclassified reports significantly drops, signifying the potential for more complete reasoning in the following years.

Discussion on the results: In this section we sought evidence supporting the hypotheses that (a) the rate of discovery for severe vulnerabilities is decreasing, and (b) the rate of discovery for specific vulnerability types is decreasing. Although the ratio of high severity bugs compared to the total is decreasing, the absolute number of severe vulnerabilities follows a similar statistically significant increasing trend as the total number of vulnerabilities, with the positive factor that the rate of increase is smaller compared to the total number of discoveries. Regarding types, we did not find evidence of a decrease of the prevalence of any of the 3 main types (CWEs 118/664/693), however we noticed that XSS and SQLI bugs due to improper neutralization are becoming rarer. Memory bugs are still very prevalent and no maturity has been achieved in this category, even though this type of bugs is most suitable for automatic detection via fuzzers, and fuzzers like the AFL have become rather popular in recent years. Our result are in agreement with Li and Paxson's [58] vulnerability life-

time measurements in 2017, where XSS and SQLI were measured to have the shortest median life span, whereas memory issues, like buffer overflows, had a median life span around three times longer.

Fuzzing is an active topic of research but, as of now, AFL and libfuzzer[21] are the major / state-of-the-art approaches in practice. AFL, in particular is not new (although it keeps evolving). If these tools had a significant impact in comparison to manual search, one would expect a sharp increase in the vulnerability rate, followed by a decline. Chromium and OpenSSL, for example, have been primary targets for in-depth fuzzing; one would hope these efforts have a significant effect. Unfortunately, based on our data and analysis so far, we cannot
* confirm this. The promising observations about XSS and SQLI however, provide us with some hope that with constant improvements in fuzzing tools (coverage, automation) we may observe a significant effect on the rate of memory vulnerabilities in the future. More research on the effectiveness of automated tools in practice and their impact on software quality is needed.

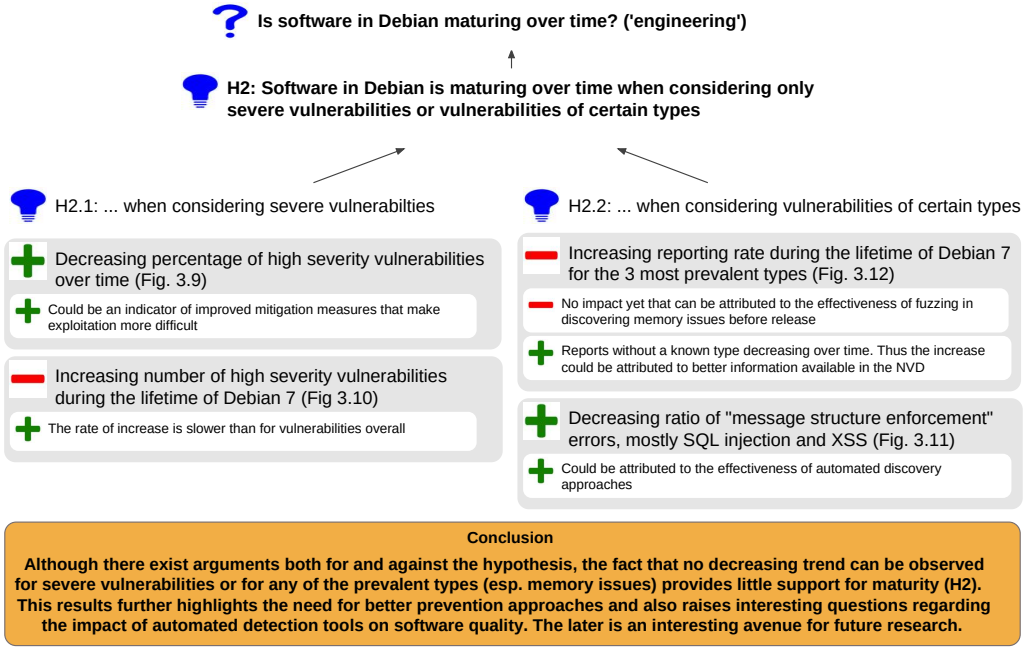
It is also valuable to compare our observations to the ones of Tan et al. [106] who investigated bug characteristics of a subset of our dataset, namely the Linux kernel, Mozilla and Apache, back in 2014. They found that semantic bugs (defined as non-memory and non-concurrency bugs according to their CWE classification) were the root cause of most (~70 %) of the vulnerabilities. We extend their results by observing trends over time and generalizing them to a complete software distribution, while getting more detailed insights by using the complete root CWE vulnerability type classification. Unfortunately, in [106] there is no trend analysis of security bug types over time to compare with our findings.

* **Summary:** In Fig. 3.15 we provide a graphical summary of the main results of this section in the form of an IBIS map. Although there were some results indicating a notion of improvement in security, following our main interpretation of maturity (i.e. decrease in the vulnerability reporting rate after a point in time), our results contradict H2.

3.6.4 *Bug bounty programs (H3)*

So far, the security quality of FLOSS does not show clear signs of improvement. Another popular argument, supported by numerous media reports, is that rewards (offered by vendors, security companies and various bug bounty programs) for the discovery of vulnerabilities are increasing. This would indicate that they are becoming harder to find and software is indeed becoming more secure. Their discovery rate may remain high because we invest more and more in searching for them. In this section, we investigate the argument mentioned above through the lens of the HackerOne bug bounty program (introduced in 3.2) via our Hypothesis 3: *Vulnerability discovery rewards for FLOSS in HackerOne are increasing*.

21 https://llvm.org/docs/LibFuzzer.html



* **Figure 3.15:** Summary of main results of Section 3.6.3 in the form of arguments pro/against the investigated hypothesis.

Table 3.4 presents a summary of the software covered by the IBB, as well as the total and maximum bounties paid in each of the projects. Apart from the projects that are named after the software components they target, there exist two more general projects. The *Data* project was launched in 2017 and rewards bugs in core infrastructure data processing libraries (e.g. curl), while the *Internet* project rewards "the most critical vulnerabilities in the Internet's history", and has famously given rewards for Shellshock ($20 000) and the Key Reinstallation Attacks ($25 000). A total of 569 reports have been awarded a

Component	Bounty #	Discl. #	Total ($)	Max ($)	Average ($)
PHP	252	236	170 500	4 000	722
Python	65	58	58 000	9 000	1 000
Data	33	18	11 000	1 000	611
Flash	69	50	175 000	10 000	3 500
NginX	4	2	6 000	3 000	3 000
Perl	12	9	7 500	1 500	833
Internet	89	26	122 000	25 000	4 692
Openssl	36	29	45 500	15 000	1 569
Apache	9	8	5 600	1 500	700
Total	**569**	**436**	**601 100**	**25 000**	**1 379**

Table 3.4: IBB dataset summary snapshot on November 2018.

total of more than $600 000 until November 2018 (counting only the 436 reports with disclosed bounty amounts), with PHP accounting for almost half of all reports, but less than 30 % of the bounties paid. Although there are interesting distinctions to be made between the projects under the IBB, for the rest of the chapter we will consider them as a uniform set of bug bounties, which to the best of our knowledge represents the most significant bug bounty program for FLOSS in existence.

First, we investigate whether security bug reports in the IBB follow the same increasing trend as they do overall. The top left part of Fig. 3.16 presents the

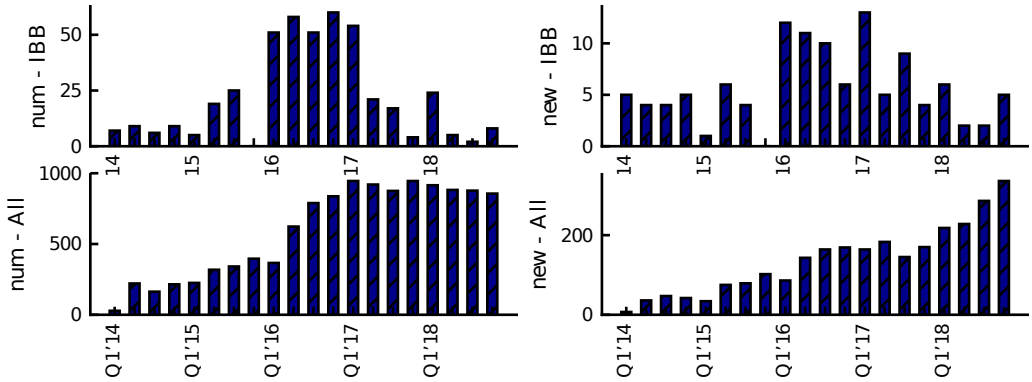


Figure 3.16: Number of claimed bounty reports (left) and new reporters (right) entering the program for IBB (top row) and the HackerOne platform overall (bottom row) over time.

number of reports that were awarded bounties by the IBB from 2014 until November 2018. Notably, the number of bounties paid shows an increasing trend until 2017 and then a decreasing trend until the end of the period in question. This behavior shows that the bug bounty program was successful (at least initially), yielding a large number of reports, and is more similar to the traditional reliability-style hardening/maturing behavior we expected to see (but did not) for vulnerability discoveries in Debian (in Sections 3.6.1 and 3.6.3). Is this a contradiction to our earlier observations? An indicator to support such an increase in quality (vulnerabilities in components included in the program are getting more difficult to find) is that the monetary amount of the awarded bounties increases over time. To test it, we look into the progression of awarded bounties over time.

Fig. 3.17 shows the trend in the amount of bounties rewarded, both for the IBB and for the whole of the HackerOne platform. The right side of the figure considers only high and critical severity bugs as classified by the bounty project's security teams (not CVSS scores – but of similar nature). Since only a few IBB bugs have been classified as either highly or critically severe, the top right box plot consists of only a few points. In general, Fig. 3.17 shows no increase in the level of the bounties rewarded in any of the 4 cases. On the contrary, the mean and quartiles of the rewards are stable over time in all cases. In combination with Fig. 3.16 and our previous observations, this points to the fact that the decrease in the number of bounties paid in the IBB may be attributed to the decrease of the attractiveness of the program in comparison to

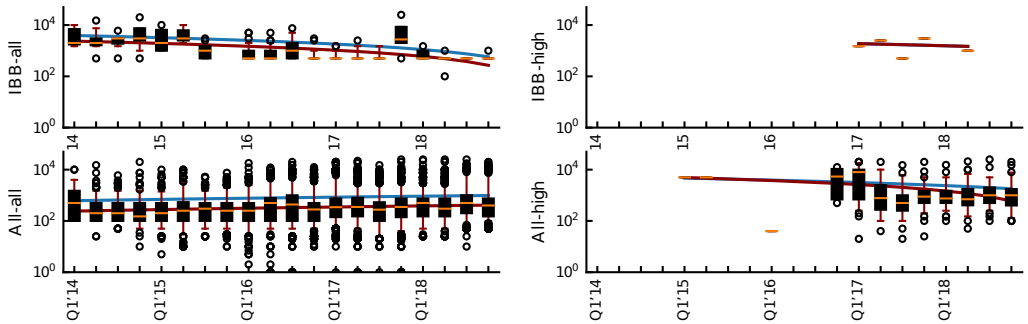


Figure 3.17: (5-95%) box plot of USD paid over time for programs in the IBB (top) and all programs in HackerOne (bottom). A further distinction is made between vulnerabilities of any severity (left) and only high/critical (right) severity vulnerabilities. Trend lines for the average (blue) and median (dark red). The only significant OLS trend comes from the top left plot concerning all the bugs reported in the IBB: a statistically significant decrease of the average as well as the median bounty. Detailed statistical test results can be found in Appendix B.2.

other programs that have entered the platform, since bugs outside the platform continue getting reported at a non-negligible rate.

Upon closer re-inspection of Fig. 3.16, we can see that the overall rate of reporting in the HackerOne platform (bottom left) is almost stable over time from the start of 2017, and this correlates with an increasing incoming flow of new reporters over time, as shown in the right part of the figure. For the IBB, the spikes in reports correlate with the introduction of new reporters in the program (rather than older reporters finding additional vulnerabilities). This is an indicator that the continued attractiveness of a program is important for its success (rather than its attractiveness at its launch).

From the above, we cannot rule out that the IBB may not have a significant market share of the hacker effort of the platform anymore, and this is the reason its effectiveness is decreasing over time, although initially being successful. We note here that an external factor that could lead to non-increasing bounties could be a lack of interest due to a declining user base for IBB software. It is difficult to estimate the user base of non web-facing software accurately, however different measurement reports point to a significant user base for software in the IBB. More specifically, measurement reports on web servers[22](Apache and nginx two leading choices), back-end programming languages[23] ("PHP is used by 79% of all websites whose server-side programming language we know"), programming languages in general[24] (Python is the most popular language with nearly 30% share), and cryptographic libraries [76] (openssl is dominant) show dominant market shares for software in the IBB at the time of writing. Moreover, note that the report rate in the IBB does not show any correlation with updates and new releases of Debian. This further supports our claim that the relative monetary attractiveness of a program is the dominating factor in the process.

22 https://news.netcraft.com/archives/2019/04/22/april-2019-web-server-survey.html
23 https://w3techs.com/technologies/details/pl-php (November 2019)
24 http://pypl.github.io/PYPL.html

To further investigate our interpretation that the IBB declined in attractiveness over time, we looked into other aspects of the behavior of IBB reporters (people with at least one IBB report) in HackerOne. Specifically, on the left side of Fig. 3.18, we see the ratio of bounties paid to those reporters for IBB reports, over the total amount they earned in the HackerOne ecosystem over time. On the right side of the figure, we see the related quantity of the number of reports filed in the IBB, over the total amount of reports filed by those reporters over time. Both plots indicate that (except from the anomaly of zero reports in Q3'15) until 2017, IBB reports came from people that hardly reported in other programs, hence they were focused only on the IBB. After that point in time, only a small portion of the reports and associated awards were in the IBB program, rather they were in other HackerOne programs. This indicates that reporters may not have expended a large portion of their effort on the IBB after 2017, instead engaging in the program rather superficially.

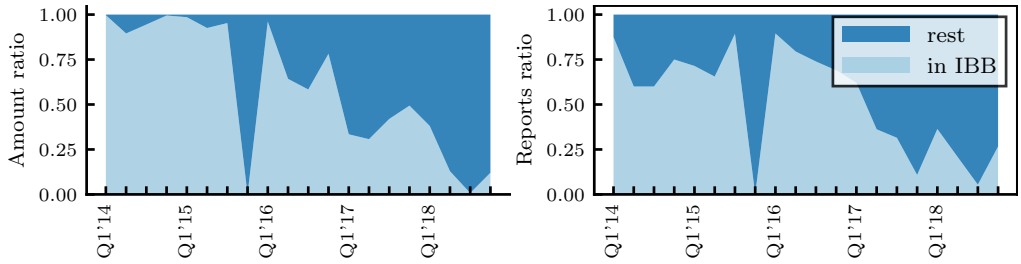


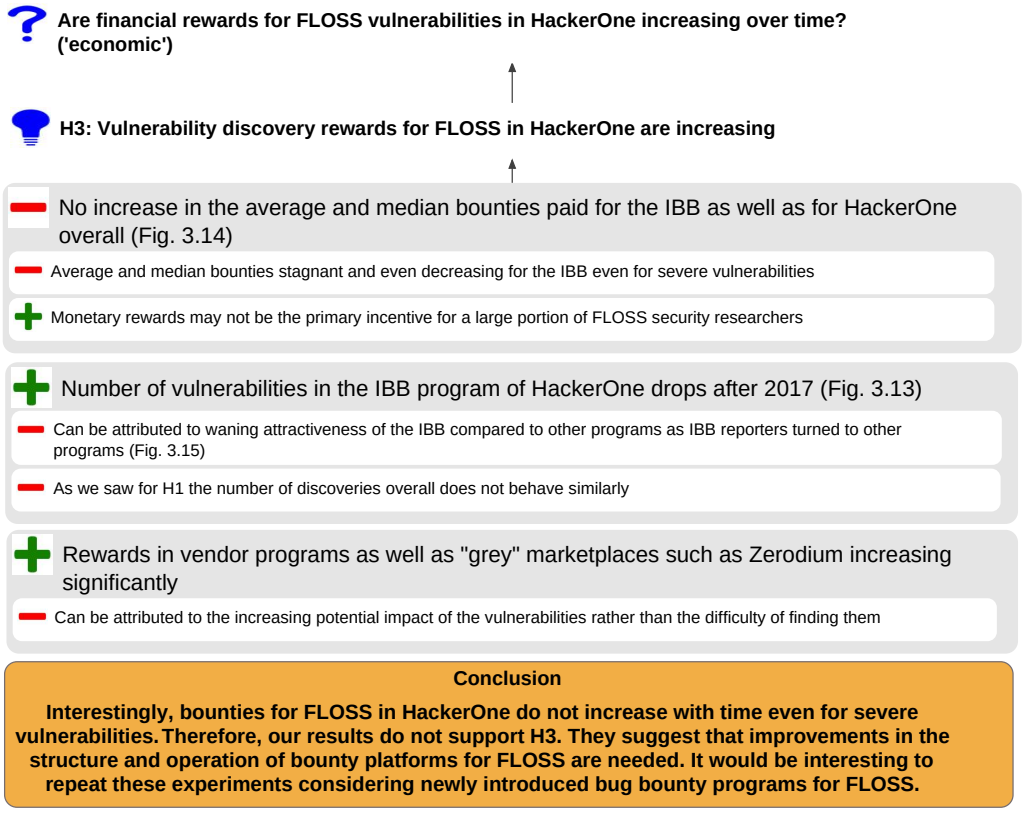
Figure 3.18: Ratio of bounty amounts (left) and number of reports (right) of IBB reporters (at least 1 IBB report at some point in time) comparing reports in the IBB program against reports for other programs in HackerOne over time.

Discussion on the results: Our results presented in this section do not support our hypothesis that bug bounty prices for FLOSS are generally increasing. It is valuable to compare our results with some recent important papers in the area. Zhao et al. [119] (2015) use the Laplace test to show that 32/49 organizations on HackerOne show a decreasing rate of vulnerability discoveries in the program and suggest that this indicates a positive effect and could be used as an indication of the web security of an organization. Considering that a significant amount of vulnerabilities affecting the software under question continue getting disclosed outside the bug bounty program, we would attribute this decrease to (a) most importantly, the limited number of hackers taking part in these programs – according to Maillart et al. [62] each hacker can only find a bounded number of bugs and each hacker's unique talents allow them to find unique bugs, and (b) a relative lack of incentives to find more difficult vulnerabilities – hackers focus on the low-hanging fruit of newly introduced programs – a claim suggested in both [62, 119]. Allodi's study of an underground black marketplace [8] shows that prices in such markets are rising over time, contrary to our results for the IBB. Thus, incentives for grey-hats to claim rewards from black marketplaces may increase. In fact, Zerodium[25], a zero-day exploit acquisition platform selling information to "a very limited number of eligible

25 https://www.zerodium.com/

customers", mainly government organizations, recently increased its rewards for an iPhone remote jailbreak up to 2 million USD. On the same platform it is advertised that a Linux PHP or OpenSSL remote code execution (RCE) can pay up to $250 000, while a Linux NginX RCE can pay up to $200 000. Naturally, these prices are a multiple of what is offered on ethical programs and by vendors themselves, since many hackers would be reluctant to sell to undisclosed government organizations. The legitimacy and legal implications of such programs (as Zerodium) is an issue that has not been discussed enough in the community.

A bright spot comes from the fact that a considerable part of the FLOSS community may be considered altruistic and/or content with "swag"/reputation rewards for discovering vulnerabilities, and therefore the collective effort expended in the vulnerability discovery process in FLOSS cannot be purely evaluated by the monetary rewards offered in bounty programs such as HackerOne. In general, we can say that bugs may indeed become more sparse inside a bug bounty program, however this can be largely attributed to the limitations of the program participants and may not be safely generalizable to claims about the overall vulnerability landscape. Overall, the impact of bug bounty programs, like the IBB is inconclusive at best and warrants further investigation. Furthermore, the motivations of reporters, especially in FLOSS, are still not well understood and also warrant further investigation.



* **Figure 3.19:** Summary of main results of Section 3.6.4 in the form of arguments pro/against the investigated hypothesis.

* **Summary:** In Fig. 3.19 we provide a graphical summary of the main results of this section in the form of an IBIS map. Note that the two main positive points in the map do not directly relate to H3 but we map them nonetheless as they are arguments that support an increase in the difficulty of finding vulnerabilities that may be relevant to the reader. Overall, our results contradict H3.

3.6.5 *Summary of main findings*

We conclude this section with a compact summary of our main findings in this chapter.

Summary of main findings

- *H1: Software in Debian is maturing over time when considering all vulnerabilities equally (Section 3.6.2):* We found no clear signs of maturing behavior (i.e. decrease in the vulnerability rate as expected by standard software reliability models) when looking into the whole Debian distribution, even when considering a single stable release over its entire lifetime. More specifically, for popular packages that underwent major updates, although a maturing behavior was observed until the next version is released, the introduction of the new version caused a surge in the vulnerability rate of the older version, indicating that maturity does not necessarily come with time.
- *H2: Software in Debian is maturing over time when considering only severe vulnerabilities or vulnerabilities of certain types (Section 3.6.3):* H2.1 (severe): Although the ratio of high-severity vulnerabilities compared to the total is dropping, their absolute number does not show a sign of decrease. Hence, no maturity (in the sense of a decrease in the rate at a certain point) can be claimed. H2.2 (types): Bug type ratio also appears stable over time, with memory indexing (CWE-118) and semantic resource control (CWE-664) bugs accounting for more than half of all vulnerabilities in recent years. Again no maturing behavior is observed for any of the main vulnerability types. Tools and automated detection methods targeting memory vulnerabilities don't seem to contribute to maturity until now, although a decrease in the prevalence of XSS and SQLI bugs show that automated approaches may indeed potentially contribute to improving quality.
- *H3: Vulnerability discovery rewards for FLOSS in HackerOne are rising (Section 3.6.4):* Our investigation of the HackerOne bug bounty platform showed that there is no increase in the rewards paid, even when considering only high severity vulnerabilities of popular FLOSS. Interestingly, the (average and median) bounties paid on the platform overall, even when considering proprietary programs, have remained stagnant over time. Furthermore, the number of bugs found in the program showed a decreasing trend, showcasing: (a) its effectiveness in the initial stages,

and (b) its relative ineffectiveness in the long term (considering bugs were found at a non-negligible rate outside the program and bounties did not increase). Our results and their contrast to recent reports of huge rewards offered for zero-days by offensive-oriented buyers provide interesting points of discussion for the community.

3.7 IMPLICATIONS AND DISCUSSION

In this section we highlight the implications of the insights gained from our detailed and large-scale investigation into the vulnerability landscape of open source software. These affect both development, distribution and testing procedures and guidelines, as well as tools, bug-finding incentives and security metrics.

Need for improved procedures.

– Longer-term support: Our measurements point out that the current duration of long-term/stable branches may not be enough to observe a maturing behavior. The maintenance of longer-term branches may be required for situations where security is an important factor.

– Stricter application of coding guidelines and testing strategies: Memory errors, like buffer overflows, continue to dominate the landscape. Further education of developers on memory issues and requirements to strictly follow guidelines could go a long way towards improving the situation. Furthermore, the improvement and deployment of development-time testing, e.g. commit-based static analysis methods like VCCFinder [88], that can be readily incorporated in development processes, should be pursued.

– Threat indicator sharing: Considering that measurements have shown that vulnerabilities have been used for zero-day attacks in the wild and have remained undiscovered for extended periods of time [15], it is valuable for organizations to share information regarding attacks with each other. Therefore, effective "real-time" information sharing platforms are required (in addition to detailed and up-to-date collections of known vulnerabilities such as the CVE/NVD). Efforts such as MISP [111] are encouraging and should be pursued further.

*

Longer-term security support may be needed in order to have releases with observable maturity. Improvements in development practices (esp. regarding memory issues) are required.

Need for new detection tools and improvements. More and better ways of finding software bugs during all phases of the software lifecycle (especially in the "testing" phase of the Debian release cycle) are needed. Tools like the kernel fuzzer syzkaller[26] paired with automatic continuous fuzzing of kernel branches (syzbot) are steps in the right direction. Furthermore, Google's re-

26 https://github.com/google/syzkaller

cently launched OSS-Fuzz project[27] is an interesting positive initiative and may produce measurable positive results in the near future. Given that memory bugs are still a large source of vulnerabilities broader and yet faster runtime memory error detectors like ASan (AddressSanitizer [101]), as well as detectors for other dangerous behavior (e.g. UBSan[28] for undefined behavior), are needed.

*

> Continuous fuzzing should be applied to an increasing number of projects. Novel fuzzing approaches (fuzzing different interfaces, new sanitizers, better coverage) should be pursued. Further and updated studies to assess and observe the impact of such tools on software maturity (potentially utilizing the platform and methods presented in this chapter) should accompany these efforts.

Need for more attractive bug-bounty programs for FLOSS. Our results showcase that bug bounty programs can be effective, however increasing prices may be required to guarantee their long-term effectiveness. The Internet Bug Bounty (IBB) program is a positive initiative that has led to the discovery of many vulnerabilities in widely used FLOSS projects. The community should look to increase the attractiveness of FLOSS bug-bounty programs in order to reap long-term benefits, and not only "low-hanging fruit". The monetary gap between bounties paid to white-hat hackers in comparison to "grey" marketplaces, and the effects of this gap, should also be seriously discussed.

> Bug bounty programs for basic FLOSS component should be discussed both withing the community as well as with policy-makers.

Need for more expressive security metrics and continuous measurement. In our study, similarly to the vast majority of similar studies in the past, we have focused on trends and attributes of disclosed vulnerabilities. More accurate and expressive metrics will further enhance our understanding of the problem and help set our priorities, and therefore progress in this area is critical. Such advancements can also enable more effective security assurance. For example, assessing the effort that was required to discover vulnerabilities (measuring the difficulty to find them instead of their number), would better capture the security quality of software. To the best of our knowledge this is an open problem. Furthermore, studies investigating the life span of vulnerabilities (e.g. [58, 84]) are either based on manual effort to link vulnerability reports to the commit the vulnerable code was introduced in, or on heuristics with limited accuracy. We visit this issue in Chapter 4. Another problem of most empirical measurement studies is that they analyze a snapshot of data at some point in time. Given that the security landscape is changing at a high rate, we need studies that are aimed at continuous measurement and plug-and-play reproducibility over time.

27 https://google.github.io/oss-fuzz/
28 https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html

> Overall, the number of discovered vulnerabilities, i.e. the vulnerability discovery rate, does not convey information regarding the security quality of software, especially when viewed in isolation. Novel metrics capturing other aspects of the process are required.

Need for more effective mitigation measures. Since vulnerabilities are seemingly not rapidly depleted, continued focus on developing and deploying mitigation measures is crucial. Software countermeasures, like control-flow integrity or sandboxing of components/libraries, coupled with hardware (CPU) features like NX/XD bits[29], SMEP[30]/SMAP[31]/CET[32], as well as recent low-overhead isolation techniques [107] limit the effectiveness of a range of vulnerabilities that could lead to control-flow hijacking exploits, and make the development of such exploits more difficult.

> Since a steady stream of vulnerabilities are being discovered, hardware/-software mitigation measures that make the exploitation of vulnerability *classes* more difficult are of critical importance.

* **Need for security by design.** Since finding and patching vulnerabilities seems to have limited influence in the overall quality of software, we should focus a larger part of our efforts in ensuring that vulnerabilities (at least of certain classes) do not make it into the code in the first place. A promising approach is programming languages designed for safety, such as Rust[33]. The fact that the vast amount of systems code is written in "traditional" programming languages that are not designed with safety in mind (e.g. C/C++) means that such a countermeasure is more long-term oriented than other mitigations mentioned above. However, we stress that such approaches of "security by design" that impede the introduction of large classes of vulnerabilities are critical in improving security and are a prerequisite for the effective utilization of ever-improving automated vulnerability detection approaches (fuzzing, symbolic execution etc.). The latter is highlighted by the fact that recent improvements in detection approaches have led to a large amount of automated reports that require an ever increasing—and often impractical—amount of effort to be manually inspected and addressed (see e.g. the more than 10 000 reports of syzkaller for the Linux kernel[34]).

> Approaches that achieve some notion of "security by design", like memory-safe programming languages, seem necessary if we want to improve software security.

29 no execute / execute disable bits to mark areas of memory as non-executable.
30 Supervisor Mode Execution Prevention
31 Supervisor Mode Access Prevention
32 Control-flow Enforcement Technology
33 https://www.rust-lang.org/
34 https://syzkaller.appspot.com/upstream

3.8 THREATS TO VALIDITY

In this section we systematically go over the threats to the validity of the study presented in this chapter following the well-established guidelines of Runeson and Höst [96][35].

3.8.1 *Threats to construct validity*

Why not density? Some studies have looked into the vulnerability rate of software per X lines of code (vulnerability density). However, this approach does not fit our goal, as we investigate whether the quality of the stable versions of software—w.r.t. security bugs—is increasing, or whether vulnerabilities are so abundant that we are not finding enough of them in a reasonable amount of time to make a practical difference (as our findings suggest). Additionally, due to the great range of programming languages employed throughout the packages of Debian, this approach would not have yielded meaningful results in our dataset. Interestingly, our findings suggest that the vulnerability density count, when considering successive releases of a software component may actually be misleading. This can be attributed to the vulnerability density dropping upon introduction of new features during the lifetime of a software release, while the absolute number of vulnerabilities may be increasing. Given our observation (Section 3.6.2) that most vulnerabilities of PHP and OpenJDK are located in components inherited by successive releases (can be considered "core" components), an observed drop in the vulnerability density rate would convey a false feeling of increasing quality.

Employed Metrics. The question here is what to measure: the quality of software development or the quality of validation/detection/testing? The two are clearly related, yet "software quality" is difficult to measure objectively. Other important metrics for software security are the speed, consistency and reach of patch distribution. In this chapter, we focus on the rate of vulnerabilities discovered over time. We observe that according to this metric, the procedures employed in development and validation of popular and widely deployed software components is apparently not effective in reducing the rate of vulnerabilities found over time. This may be an effect of the employed development model, constantly improving discovery tools and more effort and skills in the community. However, the question remains: how can there be such a constant or even increasing discovery rate over significant time frames, and what can we do to improve this result?

Security architectures. Modern software protection and isolation technology can prevent vulnerabilities from interfering with a particular platform or transaction, or make it hard to weaponize or scale a software exploit. However, even mitigated vulnerabilities are typically still reported with a corresponding reduced CVSS severity rating, since they may lead to attacks in other contexts

35 Runeson and Höst classify aspects of validity and threats in four classes: *Construct*, *Internal*, *External*, and *Reliability*.

and use-cases, or in combination with other vulnerabilities. In that sense, our dataset reflects the effects of inherent software protections that objectively affect or prevent a potential software vulnerability, but not situational/non-standard measures that address the problem only in specific platforms or use-cases. Since our goal is to analyze trends in software vulnerabilities and not a particular platform/use-case, we believe this is a fair representation of deployed mitigation strategies.

3.8.2 *Threats to internal validity*

Data quality and availability. The challenges and arising biases when performing studies with data from vulnerability databases such as the NVD were covered in Section 2.4. Here we present a summary of the issues most relevant to the analysis of this chapter and how their effect on the conclusions of the study was mitigated.

- As evident by our observations in Section 3.6.3, the NVD classification of vulnerabilities does not closely follow the proposed CWE directives, although in recent years this situation has improved. Additionally, some CWE leaf nodes have multiple parents, and some CWEs have hardly any differences between them. Therefore, we refrain from making strong claims about vulnerability type trends concerning Fig. 3.14, erring on the side of caution. Fine-grained analysis of type trends should be performed in the near future.

- The NVD is known to contain inaccuracies regarding the vulnerable program versions, e.g. as documented in Nappa et al. [74] (inducing *measurement bias*). We expect these inaccuracies not to affect our measurements, since we used the Debian Security Advisories as the root of our analysis, meaning we considered only those vulnerabilities which were recognized by the Debian Security Team to affect the package versions included in the stable release at any point in time. Of course, we cannot rule out that there exist mistakes in the Debian Security Advisories or in other fields of the NVD entries (e.g. severity, type), since these are products of manual work and may include subjective judgment. What we achieved with our technique is to avoid the known version-related pitfalls of the NVD, as well as additional bias introduced by different reporting strategies, by only considering vulnerabilities that had related Debian Security Advisories.

- Another potential source of bias in our measurements is the practice of silently patching security vulnerabilities (especially for projects that have automated patch deployment processes) or patching vulnerabilities without assigning CVE identifiers (inducing *publication bias*). For the former, although we cannot rule out that vulnerabilities have been silently patched in projects distributed in Debian, we consider our Debian dataset less affected by such bias in comparison to previous studies, as the Debian community is a fierce advocate of full disclosure and the release cycle of Debian that focuses on stable releases goes against automated updates whose content is not specified

in detail. Regarding vulnerabilities that are not assigned CVEs, collecting and including them in large-scale studies (whole distributions) seems impractical, however taking them into account when considering a smaller set of software would be interesting.

- Concerning bug bounties, we only investigated the publicly visible part of the HackerOne platform, and results may vary when considering the large number of reports that are classified. Especially the apparent huge difference in the amount of bounties offered by FLOSS on HackerOne, in comparison to big companies, e.g. Google, needs to be further investigated.

* **Dependence of discoveries.** As Ozment has pointed out in earlier work [85], vulnerability discovery events may be dependent. The discovery of a new vulnerability type, the introduction of a new automated tool, or a shift of focus of an individual (or team) on a target, can result on a number of vulnerabilities (e.g. as counted by their CVE identifiers) being discovered and disclosed at the same time (inducing *abstraction bias*). In such cases Ozment argues that these vulnerabilities should count as a singular detection event. Identifying such dependencies in the data on a large scale is understandably challenging and subject to different interpretations (e.g. one could assume that there exists a dependence relation when the same individual or team reports a batch of vulnerabilities of the same type, while someone else could assume that any vulnerabilities of the same type disclosed together or within an arbitrarily short amount of time should be considered dependent). While such batches of dependent discoveries can have a significantly detrimental effect on the fit of vulnerability discovery models, we expect their effect on the study of long-term trends to be less pronounced. Nevertheless, we acknowledge dependent discoveries as a potential limitation of our study.

3.8.3 *Threats to external validity – Generalization*

In this study, we cover a wide range of software, based on analysis of automatically collected publicly available data. Naturally, our results hold for the specific software components under consideration and no general validity is claimed. However, as stated in Section 3.2 we expect results derived for packages in Debian to be representative of other Linux distributions and a good (best-case w.r.t. maturity) sample of the state of security of popular FLOSS programs. There are many more aspects and hypotheses that could be investigated, as well as many more interesting targets. We encourage researchers to validate our results and explore further ideas using our published framework *DVAF* (see Section 3.5).

3.8.4 *Threats to reliability*

* We did not identify an threats to the reliability of the study. The publicly available code for the data collection and analysis enables reproducibility of the results.

3.9 CONCLUSION

Regarding our main question, we conclude that there is no maturing effect evident for the security of FLOSS. An interpretation could be that the current practice of vulnerability discovery is similar to "scratching off the tip of an iceberg": it rises up a little[36], but we (developers and the security community) are not making any visible progress. Our analysis is, to the best of our knowledge, the first to address the issue looking into such a large variety of software (whole Debian distribution), spanning multiple versions.

* However, not all is bleak. The community is making impressive effort in producing new fuzzing and static analysis tools, in addition to hardware security features (to make exploitation more difficult), while vulnerability reporting practices are showing signs of improvement. That being said, the need for better metrics and measurement methods, as well as studies going further than the measurement of the vulnerability discovery rate – since this rate does not seem to produce meaningful insights – are at an all-time high. Specifically, two main observations made in this chapter (see also Section 3.7) highlight two aspects of the vulnerability discovery process that appear promising in providing indications regarding the quality of security (both comparative as well as longitudinal) of a software project:

  (a.) *Vulnerability lifetimes.* Vulnerabilities seem to remain in the code for extended periods of time; measuring how long vulnerabilities live in different codebases can provide indications regarding the quality of the development and vulnerability finding processes.

  (b.) *Expended effort.* Human effort appears to be the leading factor affecting the vulnerability discovery rate; assessing the amount of effort expended in the vulnerability discovery process would allow us to "normalize" the amount of vulnerability discoveries, e.g. per a unit of effort, and therefore can help us compare between projects as well as more accurately assess the evolution of security over time.

* The two points above coincide with the second and third research questions already presented in Section 1.3, which we investigate in the following chapters. Observations presented in this chapter help the reader now better understand the motivation behind these research questions.

*

Key takeaways

There is no clearly observable decreasing trend in the vulnerability discovery rate of stable software releases even five years after becoming stable. Bug bounty prices for FLOSS do not increase either. Observations based on the vulnerability discovery rate appear to carry little meaning

36 Typically about one tenth of an iceberg's volume is above water (the "tip"), while the rest is submerged. By removing volume ("scratching") from the tip, part of the previously submerged portion will rise above the surface, so that the ratio is preserved.

w.r.t. the security quality of a software project. Further investigation of other aspects of the process is necessary. Two such aspects (vulnerability lifetimes, human aspect/effort) are investigated in the following chapters of this dissertation. The results also have important practical implications. They suggest that more focus should be put into approaches that significantly reduce the number of vulnerabilities introduced in the code, e.g. memory-safe programming languages.

4

AN EMPIRICAL STUDY ON VULNERABILITY LIFETIMES

4.1 INTRODUCTION

In the previous chapter we investigated the vulnerability discovery rates in stable releases of popular FLOSS projects. We did not observe maturing behavior (i.e. decrease) of the reporting rate even when considering the Long Term Support period of individual package versions. This result implies that vulnerabilities remain in the code for relatively long periods of time. But for how long?

In this chapter we introduce and investigate the metric of *vulnerability lifetime*. We try to answer the second research question of Section 1.3: *How long do vulnerabilities live in the code?* We do not seek to answer this question only to appease our curiosity. We rather want to identify what vulnerability lifetime can tell us about software security. Specifically, we are most interested in identifying whether the metric can provide indications regarding the relative security of different software projects, the development of security over time, and the impact of tools and practices. While the relation between vulnerability discovery rate and security may appear intuitively simple: "if the rate drops, security increases", the relation between vulnerability lifetimes and security is subtle. Shorter lifetimes could mean shorter windows of exposure, in the sense that adversaries have less time to discover vulnerabilities before they are patched – this may be interpreted as an increase in security. On the other hand, longer lifetimes could imply an increase in the quality of development, in the sense that most vulnerabilities being discovered are "ancient" and we are finding vulnerabilities faster than we are introducing them (see also Corbet's article [26]) – this can also be interpreted as an increase in security and code maturity. This subtle and elaborate relation between vulnerability lifetimes and security will be one of the main points of discussion in this chapter.

This chapter extends on the content of a research paper [2] to appear at the USENIX Security Symposium '22. The scientific contributions presented in this chapter are:

- a novel heuristic approach for accurately estimating the lifetime of CVEs without a known vulnerability introducing commit
- a rigorous validation study of the accuracy of the heuristic approach using ground truth data
- a large-scale (>5.000 CVEs) empirical study of vulnerability lifetimes in 11 popular FLOSS projects
- a critical interpretation of the presented results and a discussion of their implications

Chapter organization. This chapter is organized following the steps of the general methodology of Section 1.3. We first introduce the vulnerability lifetime metric and state the specific research questions that we later investigate (Section 4.2). Then, we provide an overview of the related work on the topic (Section 4.3) and a detailed definition of vulnerability lifetimes for software projects using version control systems (Section 4.4). Next, we describe in detail our dataset creation methodology (Section 4.5) and our novel heuristic approach for estimating vulnerability lifetimes (Section 4.6). We then present the results of our large-scale empirical study (Section 4.7) followed by a discussion on their implications (Section 4.8). Finally, we discuss some possible threats to the validity of our results (Section 4.9) before concluding the chapter (Section 4.10).

Availability

The code for the dataset collection, the estimation of vulnerability lifetimes, as well as for the analysis presented in this chapter is publicly available at `https://github.com/nikalexo/VulnerabilityLifetimes` under a free software license. We also make a snapshot of the data used in this chapter available at `https://figshare.com/s/4dd1130c336f43f6e18c`.

4.2 MOTIVATION AND RESEARCH QUESTIONS

To better convey the concept of a vulnerability's lifetime, we first briefly refresh the reader's memory of the *vulnerability lifecycle*, presented in detail in Section 2.2. A vulnerability's lifecycle, or *window of exposure* as introduced by Schneier [100] and Arbaugh et al. [13], describes the phases between the introduction of a vulnerability in the code, and the point in time when (virtually) 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 4.1. The lifecycle of a

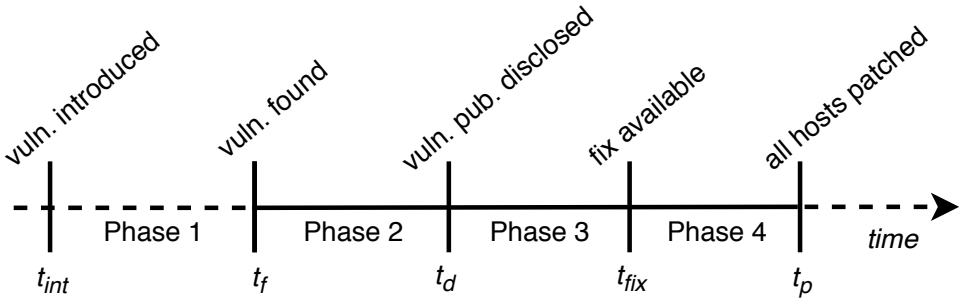


Figure 4.1: Simplified plot of a vulnerability's lifecycle. Continuous line shows period of possible exploitation.

vulnerability (or alternatively the window of exposure to a vulnerability) be-

gins 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 (most often concerning 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 [37, 74, 103], 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[1]) 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 document. This quantity can provide valuable insights regarding code maturity, can guide practical decisions, and can help us investigate fundamental questions such as: *Is the quality of software improving? Are some vulnerabilities harder to find than others? What impact do automated testing tools have on the lifetimes of vulnerabilities? Can we use lifetimes to compare software quality?*

Previous approaches towards measuring vulnerability lifetimes either relied on manual mappings of fixing commits to VCCs[25, 26, 84], which meant they were limited in scale (low number of vulnerabilities affecting one project), or used heuristics to estimate lower bounds [58, 94]. Li and Paxson [58], 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 that can be derived by its application. The main research questions we set off to answer in this chapter 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

1 The "original" codebase where development takes place. In contrast to *downstream*, which usually refers to a packaged version of the software.

software) should last, and for how long investing on dedicated long-term security support for a stable version may be necessary. Results could also be used to derive some notions of quality (of security) useful for comparisons between different projects.

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

4.3 RELATED WORK

* In this section, we place related work in the context of the content of this chapter. There exists a considerable amount of work on measuring different aspects of software security. Here we provide an overview of the work most closely related to the topic of this chapter. We set off with a brief overview of prior work on measuring different characteristics of vulnerabilities and on vulnerability discovery models (more details on these works can be found in the Related Work Section of Chapter 3). We then proceed to provide a systematic overview of notable prior work measuring the duration of different phases of the vulnerability lifecycle[2], including works that aim to measure vulnerability lifetimes, which are the ones most closely related to the contributions of this chapter.

Vulnerability characteristics and discovery rates. Studies in this field range from general measurement studies on bug characteristics [106], to studies on the vulnerability discovery rate and its trends in various software projects [3, 23, 94]. A notable strand of work focuses on vulnerability discovery models (often referred to as VDMs) 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 [5–7, 45, 49], and some as a function of expended effort, either measured as the market share of a specific product [7], or the cumulative user months estimated to have elapsed since its release [114]. 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 [3, 82, 83]. Specifically, empirical evidence (including our own – see also Chapter 3) has shown that there might be no decreasing trend in the rate of vulnerability discoveries, or even if such a trend is observed, 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. One such aspect is the time that elapses between events in a vulnerability's lifecycle, i.e. the duration of different phases of the vulnerability lifecycle. We address this body of work in the remainder of this section.

2 For a reminder on the main phases of a vulnerability's lifecycle the reader can refer to Figure 4.1 of the previous Section.

* **Duration of vulnerability lifecycle phases.** We provide an overview of the notable studies that measure the duration of different parts of the vulnerability lifecycle in Table 4.1.

Reference	Phase(s)	Details
Nappa et al. [74]	4	Patch application in Microsoft Windows hosts
Sarabi et al. [98]		
Kotzias et al. [51]		Patch application in enterprise hosts
Krebs [52, 53]	2,3	Timing of discovery, disclosure, patch and exploit availability
Frei [37, 38]		
Shahzad et al. [103]		
Bilge et al. [15]	2	Zero day attacks in the wild
Albon & Bogart. [1]		
Rescorla [94][3]	1–3	Lifetimes based on version numbers
Ozment & Schechter [84]		Lifetimes and rate of vulnerabilities in OpenBSD (focus on foundational)
Corbet [26], Cook [25]		Lifetimes of Linux vulnerabilities
Li & Paxson [58]		Quality and timing of patches – lower bound for vulnerability lifetimes

* **Table 4.1:** Overview of selected related work measuring the duration of different phases (also sub-phases and combination of phases) of the vulnerability lifecycle.

* Phase 4 of the lifecycle, namely how fast patches are applied to vulnerable systems, and how this relates to attacks in the wild, was studied by Nappa et al. [74]. They found that patching takes place at a rather slow rate, resulting in only 14% (median fraction) of hosts is patched when exploits become publicly available. Sarabi et al. [98] extended the aforementioned study by focusing on the behaviour of end users. Kotzias et al. [51] later performed a study on patching speed in enterprise hosts. They found that enterprise hosts gets patched faster than consumer hosts (comparing their results to Nappa et al.). However, there was still a significant window of exposure to most vulnerabilities in their study.
* Krebs [52, 53], Frei [37], and Shahzad et al. [103] measured characteristics of Phases 2 and 3 of the vulnerability lifecycle, namely how fast patches are made available in relation to their reporting date, the date of public disclosure of the respective vulnerabilities, and the date an exploit becomes publicly available. Shahzad et al. [103] observed that for the years 2008-2011 patches for more than 80% of total vulnerabilities in their dataset were provided before public disclosure. Phase 2 of the lifecycle, namely how long vulnerabilities remain undetected since they have been discovered by attackers and used in zero-day

3 Most of the analysis regarding lifetimes is not part of the published work provided in the reference but was part of the earlier not formally published workshop paper, which can be found here: https://infosecon.net/workshop/downloads/2004/pdf/rescorla.pdf.

attacks, was studied by Bilge et al. [15]. They found that zero day attacks on Microsoft Windows hosts lasted between 19 days and 30 months, with an average duration of 312 days.

The main topic of this chapter, *vulnerability lifetimes*, i.e. the amount of time vulnerabilities remain in the code in the (upstream) repositories of non-static projects (combined duration of Phases 1–3), has received attention, but limited progress has been made to date. A potential reason may be the difficulty of automatically assessing when a vulnerability was introduced in a codebase.

* Rescorla [94] included some rough estimates (based on often unreliable [77, 79] version information in the ICAT vulnerability database up to 2003) of vulnerability lifetimes as part of his seminal study on vulnerability depletion. He observed a mean lifetime of 2.5 years and an exponential distribution of lifetimes for the whole dataset. Ozment and Schechter [84] measured the amount of time it took for vulnerabilities of OpenBSD to be discovered and patched, focusing on *foundational* vulnerabilities (i.e. vulnerabilities that were part of the code since the first release of the software). They found that the discovery rate of foundational vulnerabilities was slowly declining over time but these vulnerabilities had a median lifetime of at least 2.6 years. Their dataset consisted of 140 vulnerabilities (87 of which were foundational) and they computed their lifetimes by manually inspecting the version control system to identify the point of introduction of each vulnerability. Their limited dataset rendered it impossible to study the development of vulnerability lifetimes over time with confidence. In a short LWN.net article, Corbet [26] presented the results of a small-scale study on 80 CVEs affecting the Linux kernel. This was followed by a blog post from Cook [25] on the amount of time 557 Linux kernel CVEs remained in the code. The latter two 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 a 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 most relevant study that investigated vulnerability lifetimes is, to the best of our knowledge, the recent work by Li and Paxson [58]. 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 a more detailed per-project analysis, our conclusions do not support their hypothesis that vulnerability lifetimes and their types are correlated. Finally and most importantly, our heuristic method for automatically estimating vulnerability lifetimes allows us to investigate with confidence crucial aspects of the problem for the first time, e.g. how lifetimes vary between different projects and how they develop over time.

4.4 VULNERABILITY LIFETIME IN VERSION CONTROL SYSTEMS

A vulnerability's *code lifetime*, or just *lifetime* for the rest of this document, 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 introduces a weakness[4], and the change in the codebase that fixes the – in the meantime discovered – weakness. 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 [67, 88] 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, for example:

- 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[5] 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[6].

- 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[7].

- 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 committed within a day[8]. 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[9].

4 What constitutes a weakness is open to definition and often also to discussion among project contributors/developers. In the empirical part of this chapter, 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 chapter, we do no differentiate between the concepts (1 CVE-ID = 1 vulnerability).

5 https://github.com/torvalds/linux/commit/b36a1552d7319bbfd5cf7f08726c23c5c66d4f73

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

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

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

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

4.4.1 *Defining a vulnerability's lifetime*

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 (as discussed in the previous paragraphs), 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 point above, 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 (*master* or *main* 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[10]. 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 becoming 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 ~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).

10 see https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git for Linux.

4.5 DATASET CREATION METHODOLOGY

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 have made a snapshot publicly available for other researchers to use[11].

4.5.1 *Mapping CVEs to their VCCs (ground truth)*

An integral part of our dataset are mappings from CVEs to their VCCs, to be used as 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 that we identified for such mappings is the Ubuntu CVE Tracker [21]. 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 excluded some of the mappings in the project from our dataset as their corresponding VCC is the first commit of the git era for the Linux kernel[12], leaving us with 824 mappings between 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 295 ground truth mappings for Chromium[13], and 74 for the Apache HTTP Server (Httpd) in the Vulnerability History Project [70]. The Vulnerability History Project is, among other things, a data source to explore the engineering failures behind vulnerabilities, and is maintained by Andy Meneely, the researcher who introduced the term Vulnerability Contributing Commit [67]. The project also contains a few additional data points for repositories other than Chromium and the Apache HTTP Server, 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 collect mappings for 1 193 CVEs to one or more VCCs, from high-quality sources. These mappings constitue our ground truth dataset.

11 https://figshare.com/s/4dd1130c336f43f6e18c

12 Linux development moved to git in 2005. The first commit of the git era for Linux https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=1da177e4c3f41524e886b7f1b8a0c1fc7321cac2.

13 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.

4.5.2 *Included projects*

For our analysis, we want a large representative sample of FLOSS projects that are compatible with our ground truth dataset 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[14] as free/libre software. Nonetheless, they are widely used in other operating systems and make up a good representation of FLOSS in general.

b. The project should have a considerable number of reported CVEs. In order to enable 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 CVEs to date.

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 for our evaluation consisting of 11 different projects of various sizes, ages and areas of application. All included projects and their respective number of CVEs and corresponding fixing commits are listed in Table 4.2. We thoroughly investigated (using a combination of automated scripts and manual effort) commit messages, bug tracking systems and NVD references, to ensure the highest possible yield of vulnerability mappings. A general description of the process follows in the next section. Details regarding the exact number of mappings found in each source for each of the included projects can be found in Appendix C.4.

4.5.3 *Linking CVEs to their fixing commits*

Our data collection process is based on information from the National Vulnerability Database (NVD) [80]. The NVD is manually curated and represents one of the largest collections of software vulnerabilities. Consequently, the NVD is frequently used in research on software security and the analysis of its data

14 https://www.debian.org/

has generated valuable insights on various topics. On the other hand, the NVD has some well-known pitfalls that the process described below tries to mitigate. This issue is further discussed in Section 4.9. For our work, we relied on the CVE-Search tool [30] to obtain a local copy of the NVD for querying and obtaining data.

For our analysis of vulnerability lifetimes, we had to link a vulnerability entry in the NVD to one or multiple commits resolving the underlying flaw in the software. These commits are referred to as *fixing commits*. In order to create the largest possible dataset 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 accepted as 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 (mentioning a CVE but not in a standard project-specific syntax), and removed 17 unjustified mappings, corrected 2, and added 2[15].

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 4.9), but are confident the errors are negligible.

4. **Third party mappings.**[16] As the information provided by linking vulnerabilities and fixing commits is useful for various purposes, fellow researchers

15 There were two occasions where a fix belonged to multiple CVEs as indicated in the commit message.

16 Note the distinction between third party mappings between CVEs and fixing commits (this paragraph), and between CVEs and VCCs (Section 4.5.1).

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 was particularly difficult. We relied on three different third party mappings: (a) data provided by the *Linux Kernel CVEs* project [61], (b) mappings that were manually curated by Piantadosi et al. [91], (c) information from the Debian Security Tracker [27]. 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.

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 4.2: 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.

These efforts resulted in a dataset of 5 914 CVEs across 11 projects that can be linked to (one or more of) their respective fixing commits. The corresponding numbers of CVEs as well as the resulting number of mappings from this process are listed in Table 4.2. 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 for which 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 regarding vulnerable products and versions (highlighted in previous research [74, 79]).

Contrary to Li and Paxson [58] we focused our efforts on a smaller set of projects, each with a significant number of NVD entries available (>100). This allowed us to study lifetimes in a per project basis. We dedicated considerable effort towards achieving as high a mapping rate (CVEs to fixing commits) as we could for the included projects. We strove to identify all potential syntaxes

for denoting bug IDs and all websites and corresponding hyperlinks belonging to the same bug tracking system. We also combined multiple of the introduced techniques. Consequently, we believe our dataset to be the most complete mapping for the included projects to date.

4.6 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 a previous work [58]. To the best of our knowledge, we are the first to evaluate this approach against ground truth data.

4.6.1 *Lifetime estimation in previous work*

Calculating the lifetime of a vulnerability is a non-trivial task, as finding VCCs can be very difficult [67]. One previously employed approach is to manually identify VCCs via thorough analysis of the fixing commits of a vulnerability [84]. While this method may yield the most accurate results, it is unsuitable for large scale studies and requires significant expertise. Therefore, Li and Paxson [58] opted for an automated approach to solve the problem. They approximated a lower bound for the lifetime of a vulnerability by appropriately using the *git blame* command. *Git blame* returns the commit that last changed a specific line in a given file (in a git repository), and so can be used to help trace a vulnerability back to its origin. Li and Paxson ran the command on each deleted or modified line of a given fixing commit; this process usually returned multiple candidate VCCs (at most one for each of the lines that were either deleted or modified). Then, they selected the most recent of the commit dates of the returned VCCs as their vulnerability introduction date, going for a lower bound approach.

Although our empirical evaluation showed that this approach is successful at calculating 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 around a year (346.88 days). Due to the large underestimation of average lifetime and large deviations of the error on different projects (see Table 4.3) we deem this approach unsuitable for our study.

A heuristic similar to Li and Paxson's was used by Perl et al. in VCCFinder [88], 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 verifying their correctness. Naturally, simply

applying an almost perfectly accurate 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 (empirical accuracy against ground truth data: ~40%). We attribute the overly optimistic evaluation of the accuracy of this heuristic by Perl et al. to the difficulty of pinpointing VCCs manually (which was their method of validating the results of the heuristic). Furthermore, we manually compared the output of their heuristic to our ground truth data for 10 randomly picked CVEs, where the two estimates differed.For 9 out of 10 cases, we found the proposed commit of our ground truth dataset to be the correct VCC, and for the remaining case we decided that both proposed commits were reasonable VCCs[17]. Our results seem sensible considering the fact that software developers put a significant amount of manual effort[18] into *regression tracking* which includes identifying VCCs. Thus, it seems likely that the accuracy of a relatively simple heuristic like this, will be limited. However, for the purpose of training a classifier that pinpoints "risky" commits with the aim of drastically reducing the search space for subsequent manual auditing, such accuracy is perfectly acceptable. Indeed, the authors of VCCFinder showed that their approach was successful in finding previously un-flagged vulnerabilities and outperformed existing approaches.

Project (CVEs)	**Li & Paxson**		**our approach 1**		**our approach 2**		**Lifetime**
	ME	St. dev	ME	St. dev	ME	St. dev	Mean
Linux (885)	-323.7	1 033.2	157.5	1 127.6	163.1	994.0	1 330.8
Chrom. (226)	-370.3	747.5	-15.5	754.1	-38.4	633.4	754.2
Httpd (60)	-599.8	1 160.0	257.4	915.8	22.4	868.9	1 890.2
All (1 171)	-346.8	993.7	129.2	1 057.9	117.0	932.5	1 248.2

Table 4.3: Comparison of heuristic performance between the lower-bound approach of Li&Paxson, "our approach 1" based on a repurposed version of the VCCFinder heuristic [88], and "our approach 2" – our optimized heuristic (weighted average). All against ground truth data and measured in days (ME: Mean Error). The last column provides the mean lifetime computed on the ground truth data.

4.6.2 *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 an efficient way to achieve this is to re-purpose the VCCFinder heuristic with some slight modifications (similar to the ones by Yang et al. [117]). 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 vulnerabil-

17 The results of the complete manual analysis of the 10 CVEs can be found in Appendix C.6.
18 https://lore.kernel.org/lkml/3519198.TemPj1OATJ@vostro.rjw.lan/
https://yarchive.net/comp/linux/regression_tracking.html

ity could have been introduced, and assigning different weights depending on how often each individual commit was blamed. Our approach is as follows[19]:

1) We use *git blame*[20] to map every "interesting" change of a fixing commit to potential VCCs. In detail, we:

- Ignore changes to test files[21], non C/C++ files, comments and empty lines.
- 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 added by the fixing commit 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. [117].

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

$$d_h = d_{ref} + \frac{1}{\sum_{i=1}^{n} b_i} \sum_{i=1}^{n} b_i (d_i - d_{ref}) \tag{4.1}$$

where b_i is determined by the number of blames the commit i received and d_i is the respective commit date[22]. For easier calculation, the dates are represented as difference in days to an arbitrary static reference date d_{ref} (January 1, 1900). According to Equation 4.1, each of the blamed commits contributes proportionally to the number of blames it received.

Using this heuristic, we decrease the mean error on our ground truth dataset (applying the heuristic on CVEs for which we have ground truth VCCs) by 66% compared to using the lower-bound blaming heuristic of Li & Paxson, effectively overestimating the actual average lifetime on average by 117 days (see Table 4.3). 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, as follows.

How many data points are needed? A standard deviation of more than 900 days means that individual measurements are generally subject to significant error and therefore not reliable for drawing conclusions. Thus, 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 4.2, we can compute the expected standard deviation of the sample mean for a

19 A preliminary version of the approach was introduced in Manuel Brack's Bachelor's thesis.

20 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*).

21 "test" as part of the file name

22 Note that in equation 4.1, operators are overloaded (to act both on dates and integers). Maybe more precise notation would be better here, however the calculation is simple.

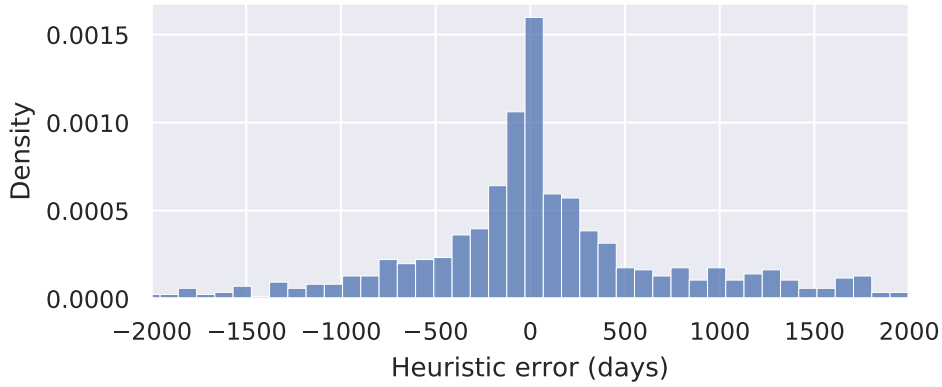


Figure 4.2: Distribution of heuristic errors in days (Excluding datapoints with no error for readability). Equally sized bins.

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%, we expect the sample mean to approximately lie in the interval [117-395, 117+395] days[23]. Making the simplification that the mean error is 0 (useful for generalization), we can say that for a confidence level of 95%, the margin of error will be $\pm$395 days for 20 samples, $\pm$250 days for 50 samples, and $\pm$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. As we will show empirically in the following paragraphs of this section, the performance of the heuristic is very good when adhering to this practice (minimum 20 samples).

Does the weighted average heuristic generalize (over time & 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 comparable for the three different projects with ground truth data available (see Table 4.3). Another source of confidence in the robustness of our heuristic is that its error is symmetrically distributed around a low mean value (that can be assumed to be zero), as can be seen in Figure 4.2. Intuitively, this means that the errors "automatically" cancel out. To understand why this is important, let us consider an alternative approach. This would be to find a suitable "perfect constant" to add to the lower

23 $\bar{x} \pm z\frac{\sigma}{\sqrt{n}}$ for $\bar{x}$=117, σ=900, n=20, and z=1.96 for a 95% confidence interval (from the normal distribution tables).

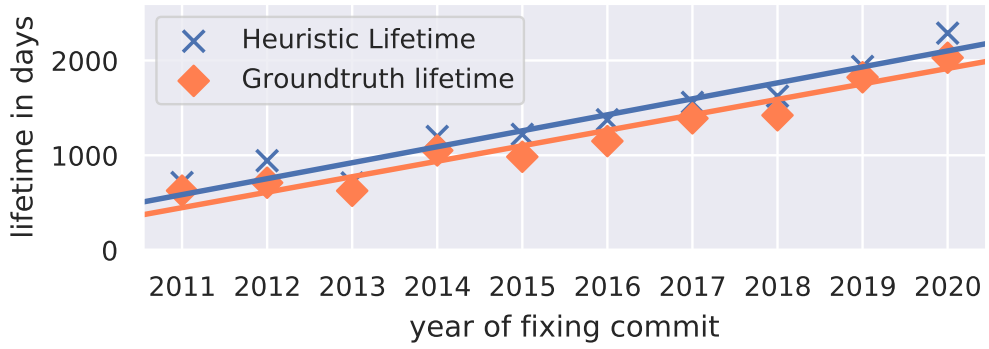
bound estimation (as computed by the Li & Paxson approach) with the goal of correcting its output to go nearer to zero. We found that depending on the data sampled, a perfect constant generally performs worse than our weighted average approach. Taking Chromium as an example, calculating a constant up 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 are split into two subsets with fixing commits before 2014, and with fixing commits in 2014 and later. Calculating a perfect constant on the first dataset and applying it to the second (by calculating the mean error of the heuristic in the first dataset and applying a fixing additive factor to the estimated mean of the second), results in an average underestimation of 228.58 days, while the weighted average heuristic underestimates the correct lifetime by only 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 projects. On the contrary, our "tuning-free" approach provides good results for all three projects, for which we possess ground truth data.

* **Can we use the heuristic to assess trends and distributions?** In the next sections we will use data points generated by the heuristic to plot and examine lifetime trends (over time) as well as the characteristics of the distribution of lifetimes. Therefore, we empirically investigate the accuracy of the heuristic in such situations.

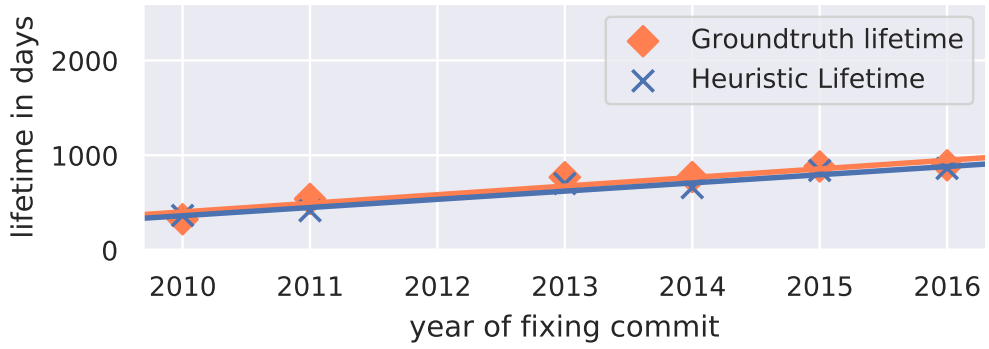
Figure 4.3a shows the ground truth lifetimes of 867 Linux kernel CVEs (for which we have ground truth data available), per year from 2011 to 2020. For the same CVEs, it also shows the estimate by our weighted average heuristic, in addition to the best linear fits for the trend in each case. Visual inspection 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. A similar conclusion can be made by looking at the corresponding plot for Chromium (Figure 4.3b). The computed gradients are 91.92 days/per year for the heuristic and 87.85 for the ground truth data in this case.

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 4.4 shows the histogram, exponential fit on the histogram[24] , and the respective Q-Q plots, for all CVEs with ground truth data and the output of the heuristic for those CVEs. From the histograms, we can see that the two distributions are similar (decreasing exponentially, similar ratio of mean to median). The Q-Q plots (comparing the distribution of the data to the theoretical distribution of the best exponential fit) are also subject to the same interpretation: excellent fit up to around 4 000 days, good fit up to around 5 000 days, and then divergence from the theoretical distribution.

24 We investigate the goodness of fit to an exponential distribution, as this is the best candidate distribution we examine later in Section 4.7.



(a) Linux



(b) Chromium

* **Figure 4.3:** Year trend comparison of ground truth and heuristic data for Linux and Chromium.

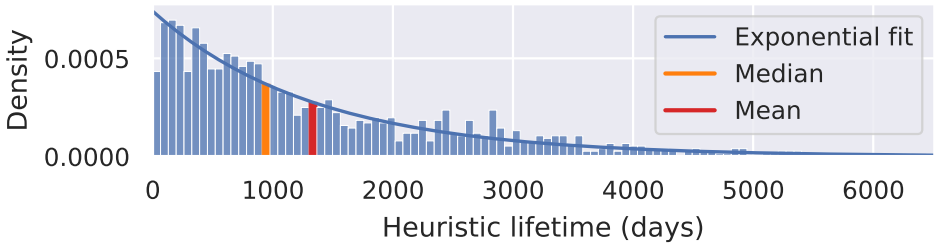
4.7 RESULTS

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[25]. For 1 171 of those data points we use ground truth data, and for the rest we use the weighted average heuristic (see Section 4.6 for details) to approximate their lifetimes.

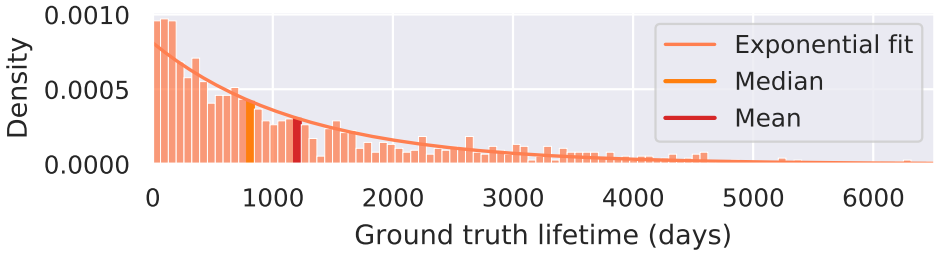
4.7.1 *General*

Table 4.4 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 [3, 25, 26, 84]. More interestingly, we observe large differences between projects (TCPDump has 4 times the average lifetime of Chromium—see Table 4.4). There can be multiple possible explanations for these differences (e.g. better security protocols, general 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 distri-

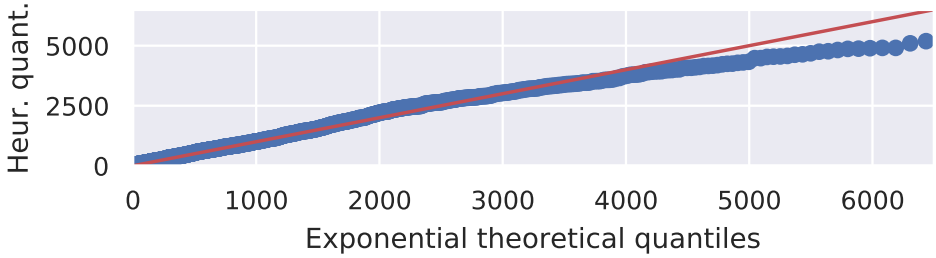
25 Due to the restrictions of the heuristic (e.g. it only considers changes in C/C++ files) the total number of CVEs with an estimated or ground truth lifetime is 5 436.



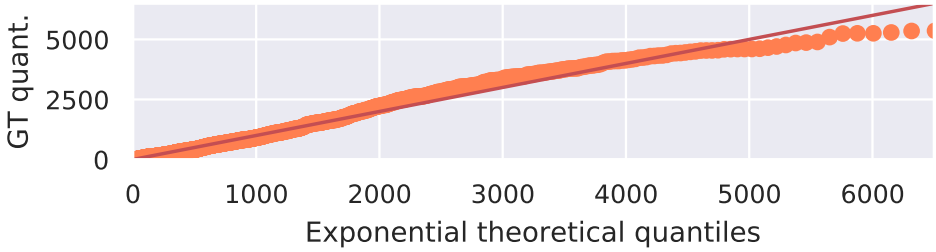
(a) Histogram for heuristic data



(b) Histogram for ground truth data



(c) Q-Q plot for heuristic data



(d) Q-Q plot for ground truth data

Figure 4.4: Histograms of lifetime distribution 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.

bution of lifetimes within a project. This issue is specifically investigated in the following subsection.

4.7.2 *Distribution*

Figure 4.5 shows the distribution of lifetimes for all CVEs, along with an exponential fit. Upon initial visual inspection of the histogram of Figure 4.5, we selected the exponential distribution as a potentially good fit to the data.

Project	Lifetime	
	Average	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 4.4: Overview of average lifetimes per project (ordered by number of CVEs)

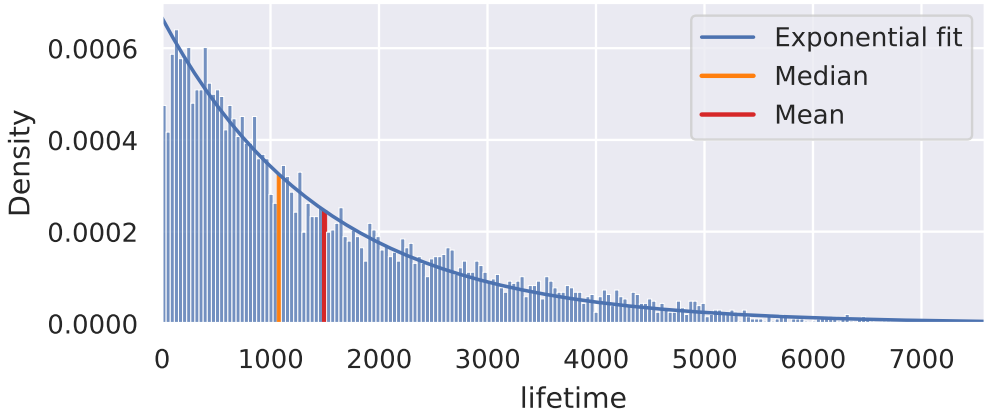


Figure 4.5: Distribution of vulnerability lifetimes. 200 equally sized bins.

* Q-Q plots [112] are commonly used for comparing empirical data to theoretical distributions (they are a more powerful technique than the common approach of comparing histograms of the two samples). Figure 4.6 shows the Q-Q plot comparing the distribution of the empirical data to the exponential fit. Interpreting the plot, we can see an excellent fit for lifetimes up to around 4 200 days that then slowly diverges. The nature of the divergence is that the empirical data are more concentrated in the region around a lifetime of 5 000 days, while the theoretical distribution expects values to increase faster. We can say that we have a good fit for lifetimes up to 5 000 days. The theoretical distribution expects 3.38% of the data points to have a lifetime of over 5 000 days,

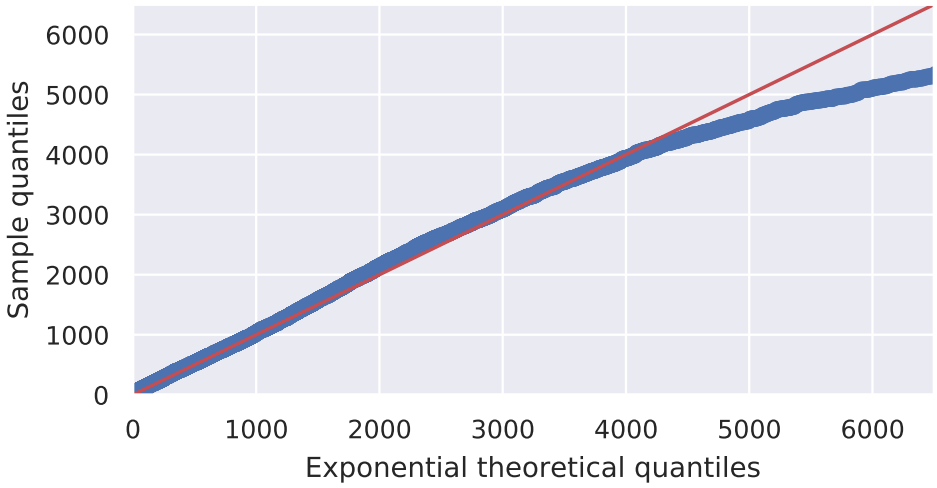


Figure 4.6: Q-Q Plot comparing the theoretical exponential distribution and our data (blue points). The fit is excellent up to a lifetime of around 4 200 days and then gradually diverges. We can say it remains a good fit up to a lifetime of around 5 000 days.

while 2% of our empirical data have a lifetime of over 5 000 days. We consider this to be reasonably small yet significant divergence, and thus assess that the theoretical distribution is a very good fit to the data up to this point (lifetimes over 4 200 or 5 000 days depending on the required accuracy of the calculation). The divergence for large lifetimes is expected, as the exponential distribution generates non-negligible mass for very large variable values (as it goes to infinity), whereas vulnerability lifetimes are naturally restricted by the age of a project.

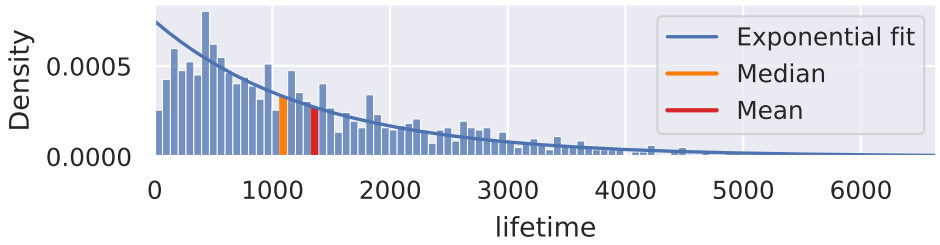
We then additionally employed the Kolmogorov-Smirnov test as described in the seminal methodology of Clauset et al. [24] to statistically compare the fit of the exponential to other candidate distributions, such as the power law or the lognormal. We found the exponential to be a statistically significant better fit. Additional details about the fitting process and further evidence supporting the goodness of fit of the exponential distribution can be found in Appendix C.3. Given all the above, the distribution of Figure 4.5 can be adequately approximated by the probability density function below[26]:

$$f(x) = \frac{1}{1501.47} e^{-\frac{1}{1501.47}x} \tag{4.2}$$

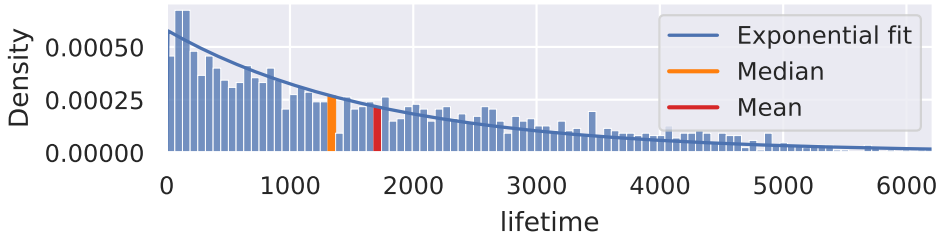
This distribution has an average value of 1 501.47 days and a median (referred to as half-life in nuclear physics) of $\ln 2 \cdot 1\,501.47 = 1\,040.74$ days. This is the amount of time required for half of the vulnerabilities to be fixed. Conversely, 63% of vulnerabilities are fixed before the average 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 – see Figure 4.7). This can also be observed in the average and median

*

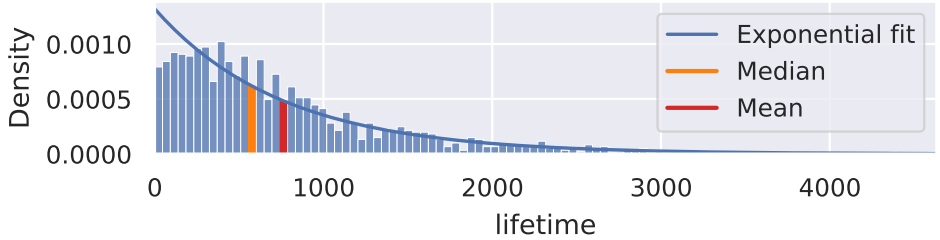
26 For most of the probability mass, except the tail (>5 000 days) as discussed above.



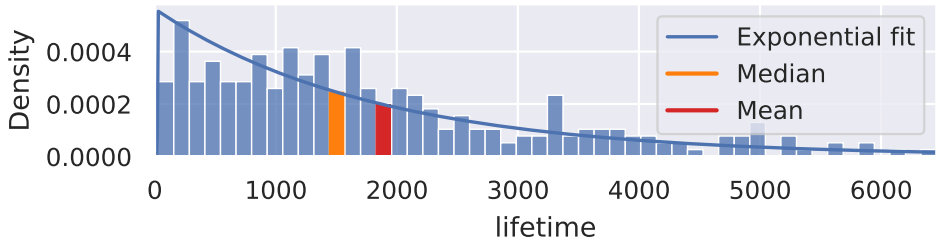
(a) Distribution of vulnerability lifetimes for Firefox



(b) Distribution of vulnerability lifetimes for Linux



(c) Distribution of vulnerability lifetimes for Chrome

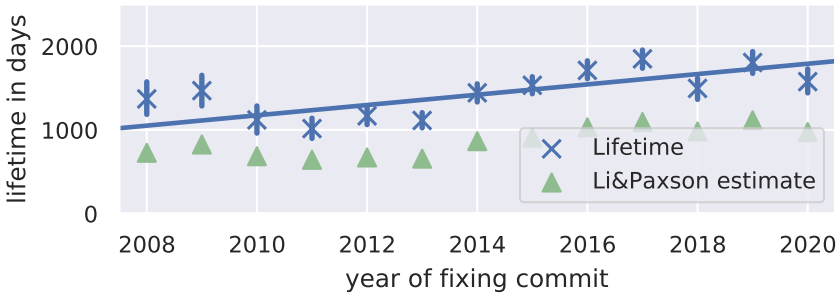


(d) Distribution of vulnerability lifetimes for Wireshark

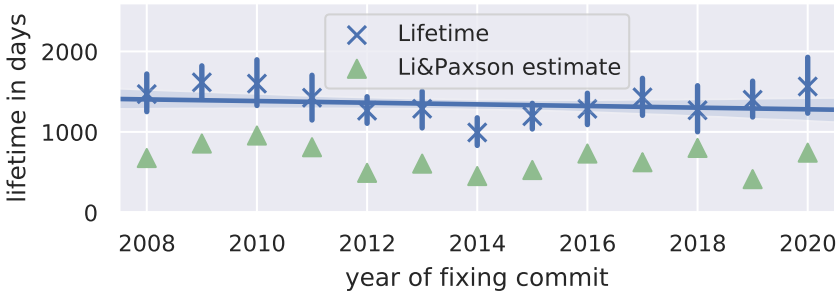
* **Figure 4.7:** Lifetime distribution per project with theoretical exponential fit (100 equally-sized bins except for the plot for Wireshark which has 50 so as not to have a significant number of empty bins).

values of Table 4.4. For most projects (especially the ones with many data points available), the median is close to ln2 (~0.69) times the average.

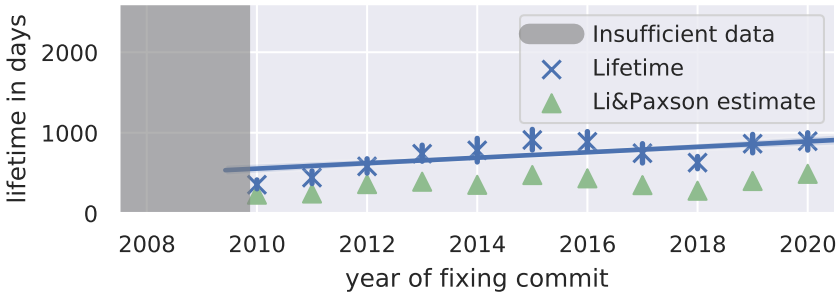
According to our analysis, for all intended purposes of this study, the empirical distribution of lifetimes in a project can be adequately approximated by an exponential distribution.



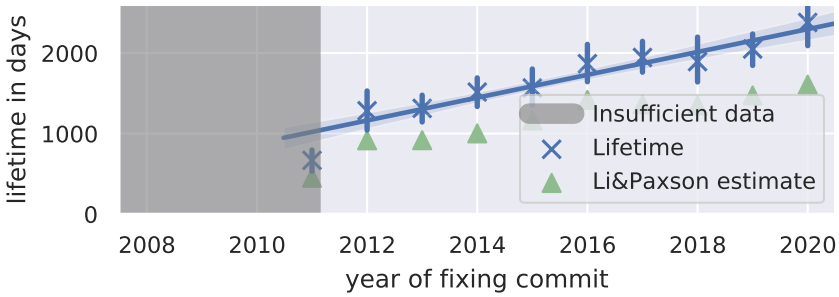
(a) All CVEs



(b) Firefox



(c) Chromium



(d) Linux

Figure 4.8: 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. Vertical error bars show confidence intervals for each year and "translucent bands around the regression line" give a confidence interval for the regression estimate. All at a 95% significance level and as computed by the seaborn python library via bootstrapping.

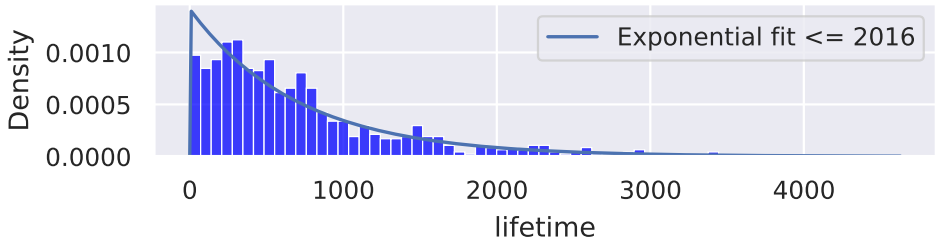
4.7.3 *Trends over time*

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 4.4) and calculated the average lifetime for each year. Figure 4.8 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.

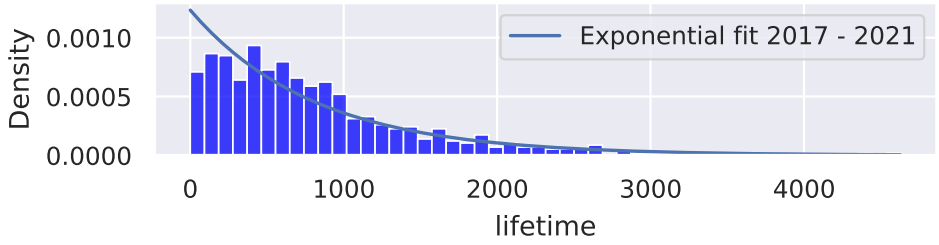
Overall (Figure 4.8a), vulnerability lifetimes show a sign of increase over the years with some fluctuation. When considering all CVEs, their average vulnerability lifetime increases by 42.78 days per year.

Considering each of the selected projects individually, for Chromium (Figure 4.8c) and Linux (Figure 4.8d) we can observe clear increasing trends, whereas for Firefox (Figure 4.8b), 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 around the linear increasing trend. 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 Appendix C.5). 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.

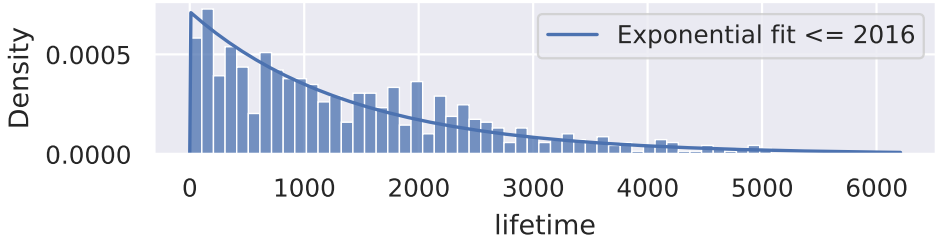
The findings above urge us to ask: *Do increasing vulnerability lifetimes mean that code quality is getting worse over time?* We came up with two possible conflicting explanations for increasing vulnerability lifetimes. The first is optimistic: we are fixing vulnerabilities faster than we are introducing 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 [26] 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 (with a decrease in the ratio of "young" vulnerabilities). As can be seen in Figures 4.9 and 4.10, 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



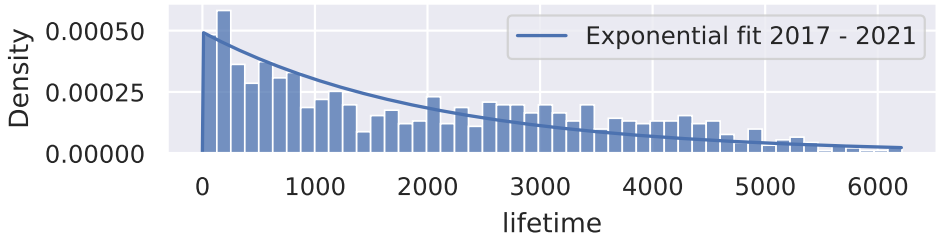
(a) Distribution of the lifetimes of 686 Chromium CVEs fixed during 2016 or earlier.



(b) Distribution of the lifetimes of 624 Chromium CVEs fixed during 2017 or later.



(c) Distribution of the lifetimes of 677 Linux CVEs fixed during 2016 or earlier.



(d) Distribution of the lifetimes of 733 Linux CVEs fixed during 2017 or later.

* **Figure 4.9:** Comparison of the evolution of the distribution of Chromium and Linux lifetimes over time.

into the relationship between vulnerability age and code age in general is required. We present this in the next section.

4.7.4 *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 considered the state of the repository on the 1st of July in that year (half-way point). Subsequently, we "blamed" (getting the point in time a line was last changed) every line in the

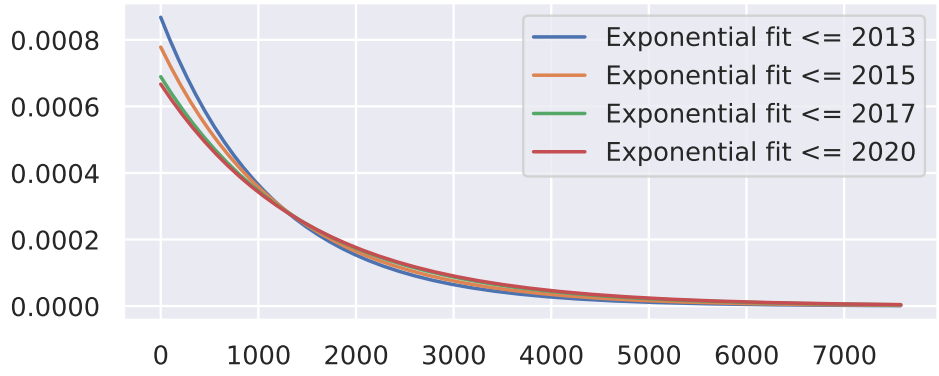


Figure 4.10: Distribution fit of lifetimes by year of fix

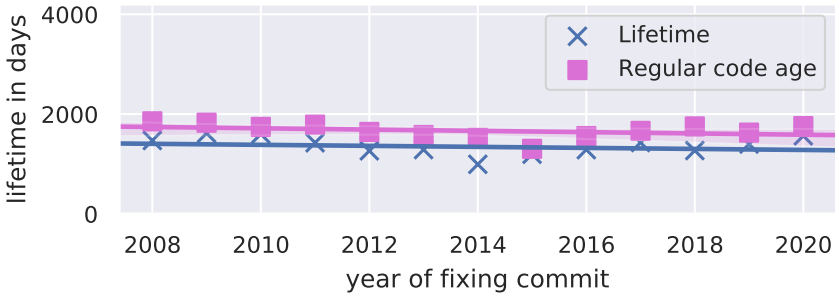
repository. We consider the time-span between the last change and the halfway point to be the *regular code age* for that line in year X. To analyze the relation between regular code age and fixed vulnerable code age (vulnerability lifetime), we calculate the average code age for each year that we have vulnerability lifetime data for (vulnerabilities fixed in that year), and plot the result in Figure 4.11 for Firefox, Chromium, Linux, and Httpd. Plots for the other projects with enough data points can be found in Appendix C.5.

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 (at the time of fix) 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 4.11b), this spread increases. These observations and their interpretation carry significant insights that we discuss in Section 4.8. 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.

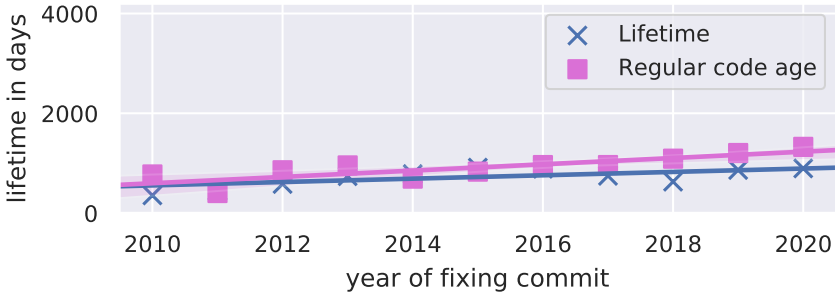
4.7.5 *Types*

CVE entries in the NVD are assigned a Common Weakness Enumeration identifier (CWE) [71] 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[27], 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.

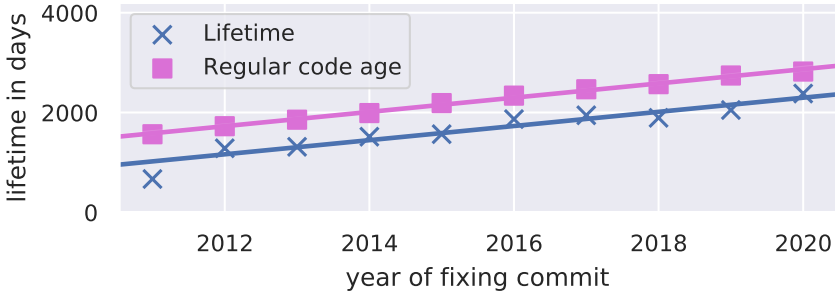
27 CWE identifiers are organized in a hierarchical structure. More general identifiers, e.g. *CWE-682: Incorrect Calculation*, have multiple more specific "children", e.g. *CWE-369: Divide By Zero*.



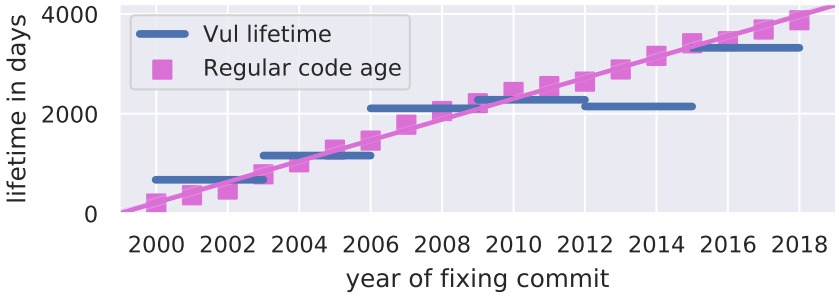
(a) Firefox



(b) Chromium



(c) Linux



(d) Httpd

Figure 4.11: 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.

Therefore, we created a mapping between CWE identifiers and 6 custom top level categories (see Table 4.5) that covers the most relevant research concepts[28].

28 This mapping was originally documented in Manuel Brack's master's thesis [18].

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 *Concurrency* 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 C.1. The mean and median vulnerability lifetime per category is shown in Table 4.5.

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

Analysis of the data for all CVEs indicates a significant[29] difference in distribution, agreeing with previous results [58].

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. Specifically, for Linux, Chromium, and Firefox, when investigating in isolation, no significant difference can be statistically observed[30]. The mean and median lifetimes per CVE category for each of these projects are given in Tables 4.6–4.8.

* To better understand the result, consider Table 4.5. Upon inspecting this table, one might claim that CVEs of category 3 (Code Development Quality) seem to be the most difficult to find, possibly due to some specific characteristic of this type. However, this claim would be far from true. Category 3 CVEs have the second lowest mean lifetime for Chromium and the lowest for Firefox, while having the second highest for Linux – but Linux CVEs have a larger lifetime irrespective of category. This can be viewed as an instance of Simpson's Paradox [17] where a statistical observation made on an aggregate population can disappear or be reversed when considering data groups individually. Overall, we observe no significant difference in the lifetimes of different vulnerability

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

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

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
4	Security Measures	736.49	545.0
1	Memory and Resource Management	770.25	618.0

Table 4.6: Vulnerability categories of Chromium, mean and median lifetimes in days

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

types. We now move on to present the results of a case study on the impact of fuzzing.

4.7.6 *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(++)[31], libFuzzer[32] and Honggfuzz[33] (syzkaller[34] for the kernel), combined with sanitizers (e.g. (K)ASan [102], MSan[35], UBSan[36]) to expose latent bugs, started being widely used for OSS in around 2016 (AFL introduced to the Linux community in 2015[37], first syzkaller talk in 2016[38], OSSFuzz launched in late 2016[39]). One may expect a measurable impact on vulnerability lifetimes by the adoption of fuzzing tools. Before approaching this question empirically, we discuss the the-

31 https://github.com/google/AFL, https://github.com/AFLplusplus/AFLplusplus
32 https://llvm.org/docs/LibFuzzer.html
33 https://github.com/google/honggfuzz
34 https://github.com/google/syzkaller
35 https://clang.llvm.org/docs/MemorySanitizer.html
36 https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html
37 https://lwn.net/Articles/657959/
38 https://github.com/google/syzkaller/blob/master/docs/talks.md
39 https://opensource.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html

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

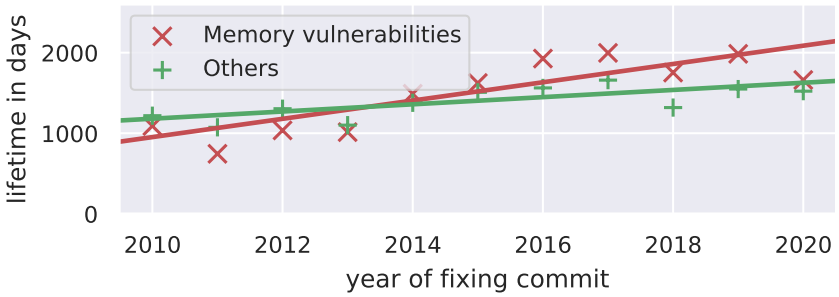
Table 4.8: Vulnerability categories for Linux, mean and median lifetime in days

oretically 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.

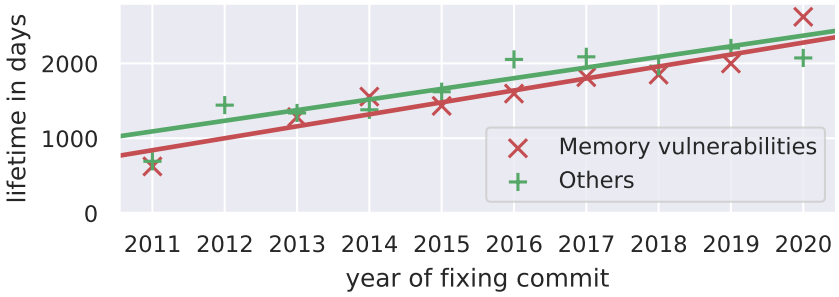
We move on to empirically approach the question. Making the simplifying assumption that memory-related bugs are the traditional and natural targets of fuzzing, we plot the trend of memory-related CVEs compared to other CVEs that fall into other categories (categories as presented in Section 4.7.5) for the whole dataset as well as for some selected projects. Figure 4.12 shows the comparison of lifetime trends between memory-related CVEs and others, for (a) all CVEs in our dataset, (b) the Linux kernel, (c) Chromium, and (d) Firefox. Overall we observe 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 in particular. Only for Chromium can we observe a possibly decreasing trend for memory-related CVEs after a spike at around 2016. This could suggest a level of maturity attained after early adoption of fuzzing tools (Google was a pioneer in such efforts). On the other hand, the comparison to the Linux kernel may be unfair since fuzzing kernel software is known to be harder than user-level applications. Future data may provide more definitive evidence.

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[40], before reporting a new batch

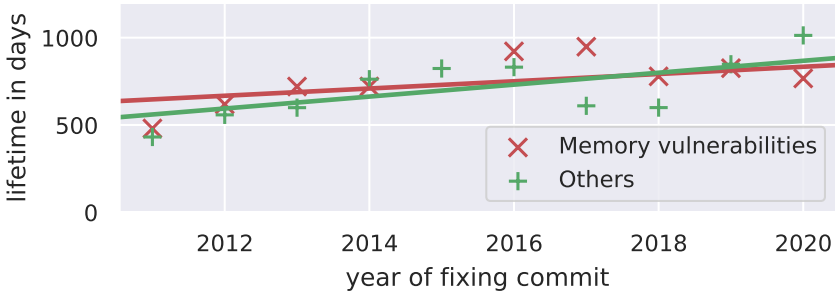
40 https://www.openwall.com/lists/oss-security/2017/12/12/7



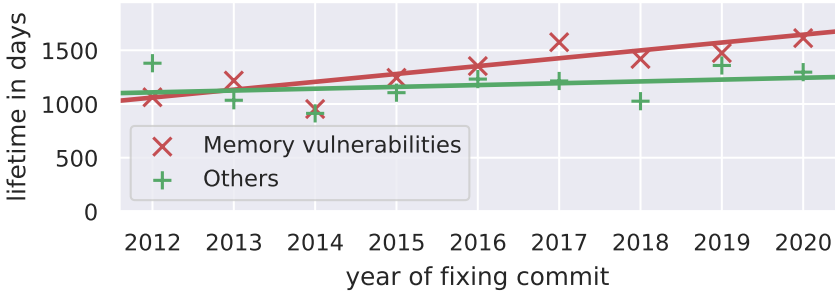
(a) All CVEs



(b) Linux Kernel



(c) Chromium



(d) Firefox

Figure 4.12: Comparison of lifetime trend in memory related vulnerabilities vs others.

in 2019[41] upon resumption of the USB fuzzing project. Moreover, Peng and Payer [87] (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

41 https://www.openwall.com/lists/oss-security/2019/08/20/2

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. Continuous fuzzing finds large numbers of bugs (e.g. >3 000 Linux bugs discovered by syzkaller and fixed[42]), in many cases fixed before the next release of a version and therefore not assigned CVEs even if they have security implications. The latter fact may explain to an extent the absence of an impact of fuzzing on vulnerability lifetimes. Furthermore, some of these bugs are not memory-related (e.g. KCSAN[43] 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.

4.7.7 *Summary of main findings*

We conclude this section with a compact summary of our main findings in this chapter.

Summary of main findings

Vulnerabilities generally remain in the code for large periods of time, varying significantly between projects (~2 years for Chromium, ~7 years for OpenSSL). Vulnerability lifetimes are distributed exponentially overall, and for each of the projects. Overall, vulnerability lifetimes grow over 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, vulnerability patches fix code that is on average younger than the code in the repository overall. This spread (between the age of vulnerable and non-vulnerable code) increases over time for some projects. Fuzzing tools do not seem to have an effect on vulnerability lifetimes, although more research is required.

42 https://syzkaller.appspot.com/upstream/fixed
43 https://github.com/google/ktsan/wiki/KCSAN

4.8 IMPLICATIONS AND DISCUSSION

Is software getting more secure over time? In Section 4.7, we presented a number of results related to this question. 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 the average age of the code at the point in time when they are fixed, and (b) for some projects this spread increases over time (see Figure 4.11).

* Observation (a) shows that vulnerable code is on average younger than the code in the repository overall at the time when the vulnerability is fixed. This observation agrees with our expectation that vulnerability finding is partly focused on newer parts of the code. It is interesting to note though that the spread between vulnerable code and non vulnerable code is small, suggesting that vulnerabilities generally still get discovered in the oldest parts of the codebase.

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 4.10 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 observations (increasing spread between vulnerable and non vulnerable code age and no change in shape of lifetime distribution) 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 expect that the vulnerability hunting process 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. Furthermore, our method for estimating vulnerability lifetimes could be used to further investigate the relation of vulnerability lifetimes and regular code age. For instance, it would be interesting to observe this relation at the point in time of vulnerability introduction (rather than fixing time as presented in this study).

> 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.

Can we compare? Meaningful quantitative security metrics are notoriously difficult to arrive at [108]. 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. For example, since the spread described above can be influenced by surges of vulnerability finding effort on newer parts of the code, to attain a more unbiased indication of maturity, we could disregard vulnerabilities with a very low lifetime (e.g. say 1 year – the head of the exponential distribution) and consider only older vulnerabilities. For these we could make the assumption that there is no age-related bias regarding the expended vulnerability finding effort and extract useful results.

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 [58]. 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 4.5 (1 760 days for all CVEs) but by far the lowest in Table 4.7 (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.

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 [120] 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 4.5). However, a significant number of vulnerabilities have large lifetimes and fuzzers keep discovering very old vulnerabilities for years (see Section 4.7.6). 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. Vulnerability lifetime can be used as a tool to quantify the impact of security tools.

* **Vulnerability lifetime and discovery models.** Modeling the discovery rate of vulnerabilities over time after the release of a software program has been the topic of extensive research (Vulnerability Discovery Models [5–7, 45, 49, 94]). These models, among others, try to determine how many vulnerabilities affecting a specific release will be discovered within a future time frame (e.g. next 5 years). The metric of vulnerability lifetime is not directly associated with the rate of discovery or the number of discoveries in general. However, it can provide useful information on how far back in the past a vulnerability originated, signalling whether it affects specific past releases. The exponential nature of the distribution of vulnerability lifetimes, in conjunction with the observation that the overall discovery rate for software projects can be quite accurately approximated as stable over relatively short periods of time (or linearly increasing over longer periods)[44], can be used to produce a simple vulnerability discovery model.
* Assume that the discovery rate (irrespective of affected version) is stable and equal to r per time unit, let's say per day. Also assume that a vulnerability is fixed instantly when discovered. As observed, the lifetime t of a vulnerability follows an exponential distribution with probability density function of the form $f(t) = \lambda e^{-\lambda t}$, where λ is the characteristic constant (mean). Then, for each day after the release of a specific version V, out of the r vulnerabilities that are discovered, a portion of them will affect V. This portion, for day i after release, is equal to the probability that a vulnerability (fixed that day) has a lifetime of i or more days. This probability is $Pr[t > i] = e^{-\lambda i}$. Then, for each day i after release, $re^{-\lambda i}$ vulnerabilities of V will have been fixed. This exponential decay is consistent with the Goel-Okumoto model for software reliability [39]

44 see e.g. www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33, www.cvedetails.com/product/3264/Mozilla-Firefox.html?vendor_id=452

as proposed by Rescorla [94]. In total, we expect to find $\int_0^\infty re^{-\lambda x}dx = \frac{r}{\lambda}$ vulnerabilities. As an example, for a specific release of the Linux kernel, if we assume a rate of 100 vulnerability discoveries per year affecting Linux in general ($r = 100/365$), and an average lifetime of 1 700 days ($\lambda = \frac{1}{1700}$), we expect to find a total of ~466 vulnerabilities affecting this specific version. Since the discovery rate and lifetime of vulnerabilities seem to develop relatively predictably over time, we can substitute the constants with relevant time-related functions and get more accurate results. Note that as long as there is a strong correlation between vulnerability lifetime and general code age, we cannot attribute the resulting exponential decay behavior of this model to a depletion of the total number of existing vulnerabilities for a release (as is the case for reliability models that exhibit similar decay), rather than to the inherent code churn.

*

Overall, vulnerability lifetime, as a metric, can support building theories regarding vulnerability depletion in specific software versions. A simple model based on our observations in this chapter agrees with the Goel-Okumoto model for software reliability.

* **Stable freeze time and long-term support.** Intuitively, the fact that vulnerability lifetimes are exponentially distributed would indicate that a relatively short stable freeze time would be enough to discover a large portion of vulnerabilities for security sensitive applications. Indeed, employing a stable freeze period of around 1 000 days for security-critical systems, would be sufficient to avoid half of the ultimately discovered vulnerabilities. However, the high mean of the distribution (often >1 500 days), along with the defender-attacker asymmetries of security (the attacker only has to find one weakness before the defender), indicate that for high levels of perceived assurance (e.g. 80% of the eventually discovered vulnerabilities have been fixed), large periods of time are needed (on average ~6 years). Also, note that according to the previous point, code churn may be the main force behind decreasing vulnerability discovery rates for older versions at the moment. Thus, there is no guarantee that a motivated attacker will not be able to discover a vulnerability affecting an older version that may never be discovered by the white-hat community. That being said, there is a sizable portion of risk in the patching process, specifically concerning the time between a fixing commit becoming available, its back-porting to the version compatible with a specific system, and the actual application of the patch [74, 85]. In that regard, a smaller number of security patches that need to be applied translates to risk reduction.

* The distribution of vulnerability lifetimes can also be used to inform decisions on end-user support agreements about products with open source components. For example, a distributor may be able to say to a customer that they offer long-term support of X years. Then, the customer would be able to estimate the expected number of remaining patches, assess cost of identifying, developing and applying them in-house, and make a decision.

*

> Overall, vulnerability lifetime can provide actionable information to guide practical decisions, e.g. regarding the duration of the stable freeze and long-term support phases, that can be especially useful for security-critical systems.

4.9 THREATS TO VALIDITY

In this section we systematically go over the most important threats to the validity of our results.

4.9.1 *Threats to construct validity*

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 (mean error of ~100 days as discussed in Section 4.6.2) to affect their validity.

4.9.2 *Threats to internal validity*

Dataset. The data in vulnerability databases often experience bias from several sources [22]. Specifically, for our study the following apply:

- 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 may be necessary to investigate how results differ when also considering such 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. We share our dataset to be used and improved by the community.

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 [85], either due to

a new class of vulnerabilities being discovered, a new tool or method 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 (Section 4.8) 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.

4.9.3 *Threats to external validity – 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.

4.9.4 *Threats to reliability*

* We did not identify any threats to the reliability of this study. The publicly available code and dataset used in the study enables reproducibility of the results.

4.10 CONCLUSION

In this chapter we introduced the metric of a *vulnerability's lifetime*. This is the amount of time a vulnerability 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 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.

* Notably, although as repeatedly stressed the task of assessing the "quality of security" of software is a highly intricate matter, vulnerability lifetimes and their relation with the average age of a codebase can provide useful relevant information. We observed that for some projects the mean lifetime of code that is fixed decreases over time compared to the age of the codebase. As discussed in Section 4.8, such a development could be an indication of increased quality, at least for some (older) parts of the code. Therefore, we consider the development of the spread between the age of vulnerable code and the age of the codebase

at the time of the fix to be a promising metric towards assessing the relative quality of software (*"comparing"*).

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 (briefly touched upon in Section 4.8), could provide interesting insights. In addition, further research on the matter from the perspective of a vulnerability's introduction date (rather than its fixing date, which was the focus of the study presented in this chapter) may provide further interesting results.

*

Key takeaways

We can compute vulnerability lifetimes automatically and accurately. We showed that vulnerability lifetimes can provide indications regarding the maturity of codebases, can help identify differences in the performance of security practices between projects, and can provide means to measure the impact of security tools. Overall, vulnerability lifetime is a promising metric towards further understanding software security.

5

AN EMPIRICAL STUDY ON VULNERABILITY REPORTERS

5.1 INTRODUCTION

In the previous chapter we investigated vulnerability lifetimes in popular FLOSS projects. We presented a novel method to automatically calculate the average lifetime of a large enough set of vulnerabilities. We used this method to perform a large scale longitudinal study on the vulnerability lifetimes of 11 FLOSS projects. We showed that vulnerability lifetime, especially in relation to the code age of a project, can provide an indicator of maturity. Furthermore, we discussed several theoretical and practical implications of the metric, showing that the metric offers a promising way towards better understanding software security. Now we continue our search for measurable indicators of quality.

In this chapter we try to measure an elusive quantity: *vulnerability-finding effort*. We try to answer the research question (third research question of Section 1.3): *How can we measure human effort in the vulnerability discovery process?* The effort expended in the vulnerability-finding process has long been considered one of the dominant factors behind the vulnerability discovery rate (see also our results from Chapter 3). Thus, in this chapter we aim to capture an aspect of this effort by analyzing the characteristics of the people who report vulnerabilities in FLOSS projects, and by investigating how these characteristics can inform us about software security quality.

This chapter extends on the content of a research paper [4] accepted for publication at the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM 2021). The scientific contributions presented in this chapter are:

- a novel methodology for creating the first (to the best of our knowledge) dataset of vulnerability reporters for FLOSS
- a large-scale (>2 000 reporting entities for >4 500 CVEs) empirical study on the characteristics of vulnerability reporters in 4 popular FLOSS projects
- a critical discussion on what the characteristics of vulnerability reporters convey about software security quality, along with other implications of our results

Chapter organization. This chapter is organized following the steps of the general methodology of Section 1.3. We first introduce the topic, provide some motivational background, and state the specific research questions that we investigate in this chapter (Section 5.2). Then, we provide an overview of the related work on the topic, especially in relation to the contributions of this chapter (Section 5.3). Next, we describe in detail our dataset creation methodology (Section 5.4). We then present the results of our large-scale empirical study (Section 5.5) followed by a discussion on their implications (Section 5.6). Finally, we

discuss some possible threats to the validity of our results (Section 5.7) before concluding the chapter (Section 5.8).

Availability

The code for the creation of the dataset and the analysis presented in this chapter is publicly available at `https://github.com/nikalexo/vulnerability_reporters` under a free software license. We also make a snapshot of the data used in this chapter available at `https://doi.org/10.6084/m9.figshare.14986830.v1`.

5.2 MOTIVATION AND RESEARCH QUESTIONS

Rather than a deficiency in functionality, a vulnerability is an unintended feature that can potentially enable attackers to compromise a system. Discovering a vulnerability involves consideration not just of what the system was intended to *do*, but what is possible to *exploit*. This attacker mindset is not ingrained in every developer, so Free/Libre and Open Source Software (FLOSS) projects must rely on the diverse skillsets of their community to both discover and responsibly disclose these vulnerabilities. Despite the great efforts of the community in investigating different aspects of the vulnerability discovery process, the "who"s of the process have not yet received enough attention, especially outside the closed environment of bug bounty programs.

* Increasing our understanding of the human factor in the vulnerability discovery process can have important implications for software security metrics. Characteristics of vulnerability reporters[1] (e.g. their number, its progression over the years, their motivations) can be used to gain insights regarding the amount of vulnerability finding effort (*community engagement*) invested into the process and its variation over time and between different projects. These insights may enable us to derive indicators of health for the security ecosystem of a project, and as a consequence, indicators for its security quality.

* To get a rough idea of how information about vulnerability reporters may be utilized as an indicator of quality, consider the following simplified scenario: "Projects A and B each had an average of 10 vulnerability reports per year for the last 5 years. All vulnerabilities of project A were reported by a single person whereas 20 different individuals with 10 different affiliations reported vulnerabilities for project B during this time". Based on the information above, one may claim that the vulnerability-finding effort that led to the discovery of the vulnerabilities of project B is greater than that of project A. This claim would imply that it was more difficult to find vulnerabilities in project B, and therefore it can be used as an indicator suggesting that project B is of higher quality than project A.

1 Although there is a slight difference between vulnerability reporters and discoverers (we discuss this in Section 5.7), for the rest of this chapter, we assume the terms reporters and discoverers to be interchangeable.

Complementary to implications regarding security metrics and measurements, increasing our understanding of the human factor in the process can have further implications on development and security practices. FLOSS project coordinators need to know their community of vulnerability reporters if they want to maintain the health of the project long-term (since this may depend on keeping them involved). FLOSS maintainers, as well, need to understand who they will be working with in the disclosure and fixing process.

Overall, the goal of this chapter is to shed light on the human aspect of the vulnerability discovery process by analyzing historical vulnerability reporting data of mature and successful open source projects, in order to: (a) identify common trends and practices that can act as benchmarks of community engagement for new and existing projects, and (b) examine whether (and to what extent) empirical metrics of community engagement can act as indicators of health and quality (of security) of software.

* **Research questions:** To approach our goal, we structure our investigation by asking the following four research questions (each accompanied by a number of more specific questions detailing the scope of the investigation):

  – **[Distribution] What is the distribution of reports among the reporters? (results in Section 5.5.1):** *Are contributions to the vulnerability discovery process (reported vulnerabilities) evenly distributed among the contributing entities (decentralized) or significantly concentrated in a small number of reporters (centralized)?* Answering this question can provide insights as to whether (a) "many eyeballs" (a reference to Linus's law [92]) is the primary contributing factor behind vulnerability reports, or (b) dedicated security resources (translating to a large number of vulnerabilities reported by a small number of entities—attributed, e.g. to the effectiveness of automated tools or the increased specialization required to discover vulnerabilities) is the primary contributing.

  – **[Temporal characteristics] How do the characteristics of reporters develop over time (results in Section 5.5.2):** *Do more vulnerabilities mean more reporters over time? For how long do reporters stay engaged?* Such temporal characteristics can provide insights regarding the relation between effort and discovery rate, as well as the development of practices over time.

  – **[Specialization] Are reporters specialized? (results in Section 5.5.3):** *Are productive reporters specialized w.r.t. projects or specific types of vulnerabilities?* If they are, then projects would not only need to attract an adequate number of reporters but also a set of reporters with a diverse set of specializations.

  – **[Motivations] What are the motivations of reporters? (results in Section 5.5.4):** We also want to explore the motivations of reporters in the projects of our study. *Does a large portion of yearly reports come from reporters that have otherwise (e.g. via code commits or non-security bug reports) contributed to the project? Are most reports coming from reporters internal to the organization behind the project? Are they employed by other organizations? Did they receive bounties?* Answers to these questions can provide insights into how established open source projects attract vulnerability reporters, which in turn can be translated to strategies for new and emerging projects.

5.3 RELATED WORK

In this section, we provide an overview of the related work on vulnerability reporters and place our contribution in perspective.

User studies on reporters. The most closely related works on vulnerability reporters are based on conducting user studies. The work of Fang and Hafiz [33, 40] is especially relevant to ours. They ran an email-questionnaire user study, collecting 127 responses from a variety of reporters of buffer overflow, SQL injection, and cross-site scripting vulnerabilities. Their study was, to the best of our knowledge, the first to target reporters of vulnerabilities. They focused on the approach, tools and techniques of reporters of different types of vulnerabilities affecting a variety of software, as well as their disclosure practices. One of the results of their user study that our analysis corroborates is that reporters seem to be specialized w.r.t. the types of vulnerabilities they report (Section 5.5.3). In their user study, a large proportion of the reporters who responded to the questionnaire, claimed to have reported a large number of vulnerabilities. Our data, on the other hand, indicates that most reporters report a small number of vulnerabilities 5.5.1). This is probably due to the inherent bias induced by the user study (it might be easier to contact a reporter with multiple reports, such a reporter may be more likely to respond to a request, etc.).

Another interesting user study was performed by Votipka et al. [110]. They used a semi-structured interview technique on a sample of 25 testers (from within a project) and white-hat hackers to investigate differences in the vulnerability finding processes of the two groups. They concluded that the approaches of the two groups differ significantly and a project would benefit from the engagement of both groups in its community. They also pointed out that hacker engagement can also be achieved via non-financial rewards and not only with traditional bug bounties. Our analysis confirms these results, as we see a strong contribution from both long-term reporters and "come-and-go" hackers, with bug bounties not necessarily being the incentive (Section 5.5.4).

* Overall, the nature of our study (data-driven, large-scale empirical study) differs fundamentally to the user studies referenced above, as the scale and completeness of our collected dataset (see Section 5.4) allows us to investigate aspects of the process that cannot be investigated via user studies (e.g. development over time, differences between projects). Furthermore, the focus of our study also differs. Previous studies focused on how vulnerabilities were discovered, while we focus on how we can attain estimates of human effort and indicators of quality.

Bug bounty programs. Most previous large-scale studies providing insights into vulnerability hunting do so from within the bounds of bug bounty programs [36, 42, 62, 73, 119]. Although we saw that bug bounty programs are important in incentivizing vulnerability reporters for some projects, our findings suggest that their effect in the FLOSS ecosystem as a whole seems to be rather limited (Section 5.5.4). Furthermore, none of these studies focused on the specific reporters, more on the structure of the bounty programs themselves.

Investigating Linus' law. Linus' law is the proposition that a large community of contributors and testers improves the quality of open source software. Eric Raymond first formulated the law as "Given enough eyeballs, all bugs are shallow" [92]. Meneely and Williams [68, 69] explored the correlation of developer collaboration metrics and discovered vulnerabilities for popular open source projects. They found that files co-developed by 2 or more independent developer groups were more likely to contain a vulnerability than files developed by collaborating contributors. Also, Linux kernel files with changes from 9 or more developers were 16 times more likely to have a vulnerability. Favato et al. [34] found that java projects with more committers on Github tended to have more bugs. These studies investigated the "developer" aspect of Linus' law. Our study, on the other hand, provides insights regarding the "bug-finding" aspect of the law.

* **Summary.** Overall and to the best of our knowledge, our study differs to prior studies on the human factor in the vulnerability discovery process in two fundamental aspects:

- (*Reach and scale*) Previous studies investigated characteristics of vulnerability reporters either via user studies (small scale) or within the bounds of bug bounty programs (capturing only a small part of the ecosystem). We aim to capture characteristics of vulnerability reporters for the whole ecosystem of popular FLOSS projects by collecting as complete as possible a dataset of reporters from a large number of sources.
- (*Goal*) Previous studies either aimed at learning more about how vulnerabilities are discovered in the wild (which tools, techniques, etc.), or how effective bug bounty programs are. Our study, on the other hand aims at producing insights regarding indicators of vulnerability-finding effort and of software quality.

5.4 DATASET CREATION METHODOLOGY

The following sections describe our methodology for constructing what is, to the best of our knowledge, the largest and most complete dataset of vulnerability reporters in existence.

We focus on four big community-driven open-source projects: the Linux kernel, the Apache HTTP Server Project, the Mozilla suite (including Firefox and Thunderbird), and the PHP interpreter. We chose these projects as they are popular community-driven open-source projects with a large number of reported vulnerabilities and a transparent process for reporting and fixing them. We considered the number of projects (four) to be a reasonable compromise between required manual effort and utility of the created dataset. In the following, we introduce a methodology to collect information regarding vulnerability reporters from a variety of sources, including the NVD, the projects' bug reporting platforms, and the projects' version control systems. We note that the process is

best-effort and may still be subject to errors, but the dataset[2], as well as all our scripts are open and publicly available under a free software license[3].

Such a dataset can be used to investigate a wide range of characteristics of the vulnerability discovery process (apart from the research questions in earlier sections). Thus, we consider the dataset collection methodology to be an indipendent contribution of general importance.

5.4.1 *Information on included projects*

Different software projects/development teams use diverse approaches regarding reporting, patching and documenting bugs. Their processes differ in the way they handle vulnerabilities.

– **Mozilla suite:** The Mozilla suite includes the code for Mozilla products such as the popular Firefox web browser and Thunderbird email client. The suite is handled as a whole since these products share a significant underlying codebase (the *Core* component). Both security and non-security-related bugs are typically reported and handled in the Mozilla Bug Tracking System which is implemented as a Bugzilla instance (although security bugs can also be sent to the security team via email). Commit messages in the repository that fix a bug should include the bug id (identification number). The Mozilla security community also publishes security advisories providing more information on the fixed security bugs[4].

– **Apache httpd:** Apache httpd is a popular web server with a market share of over 33%[5]. The Apache security policy[6] states that security bugs are to be reported in a dedicated private mailing list and are handled differently than normal bugs (which are handled via a Bugzilla bug tracking system). The Apache security team also publishes security advisories for all fixed vulnerabilities affecting released versions.

– **PHP:** PHP is the most popular server-side programming language in the web at the time of writing[7]. The standard PHP interpreter is a community-driven project written almost entirely in C using git as the version control system. Security and non-security bugs are both handled by the PHP bug-tracking system (albeit in different ways and with different priorities and privacy rules). Developers are requested to add the bug number prepended by a "#" in the commit message of the fix.

– **Linux:** The Linux kernel is arguably the most impactful community-driven project in history, thus being the prototypical example of Eric Raymond's bazaar [92]

2 https://doi.org/10.6084/m9.figshare.14986830.v1, Data are collected from publicly available sources whose purpose is to credit vulnerability reporters. Thus, the data were manifestly made public, and as such processing of the data does not infringe on the privacy rights of the involved individuals, e.g. see Article 9 of the GDPR.

3 https://github.com/nikalexo/vulnerability_reporters

4 https://www.mozilla.org/en-US/security/advisories/

5 https://w3techs.com/technologies/details/ws-apache. Market share: 33.6% on 14th May 2021.

6 http://www.apache.org/security/

7 https://w3techs.com/technologies/details/pl-php. Market share: 79.2% on 14th May 2021.

model of community-driven software development. Although most contributors to the project are no longer volunteers (since they are employed by several organizations to work on the kernel), the general concept of the bazaar, i.e. decentralized and lightly coupled development, still generally holds. Security bugs in Linux are to be reported to the kernel security team via email and are handled separately from normal bugs which are handled via a combination of (subsystem-specific and general) mailing lists and a Bugzilla instance.

5.4.2 *Data sources*

Pursuing the goals of the study requires the collection of several data points for the four FLOSS projects investigated. We proceed to document the data collection process for each data type:

1. CVE entries: We use the cve-search tool[8] to maintain a local copy of the CVE information available at the NVD.

2. Vulnerability reporters: Information on who first reported a vulnerability is not available directly in the NVD entries. Upon further investigation, we located the following sources of information about reporters:
   - For Mozilla products, relevant information is available (a) at the "Reporter" field of Mozilla Security Advisories[9], (b) at the "Reporter" field of the associated bug report.
   - For Apache httpd, relevant information is available at the "Acknowledgements" field of the Apache httpd security advisories[10].
   - For PHP, relevant information is available at the "Reporter" field of the associated bug report.
   - For the Linux kernel, relevant information is available (a) in Android security bulletins[11] (for vulnerabilities affecting a supported Android release), (b) in the description of Debian Security Advisories[12] (for vulnerabilities that affected the kernel version included in the stable Debian distribution at that point in time), (c) at the "Credit" field of the SecurityFocus database[13], as well as (d) in Ubuntu Security Notices[14] and (e) the Red Hat Linux bug tracking system[15]. The four latter sources of information (b) – (e) maintain such references in general for all projects (since they are reporting outlets for software distributions which include all 4 projects), and are therefore considered for all of them.

8 https://github.com/cve-search/cve-search
9 See Footnote 4
10 https://httpd.apache.org/security_report.html
11 https://source.android.com/security/bulletin/
12 https://www.debian.org/security/
13 https://www.securityfocus.com/
14 https://ubuntu.com/security/notices
15 https://bugzilla.redhat.com/index.cgi

Although the information of the sources above is not included in the NVD, references to these sources are often included, making the mining process easier.

3. Bug reports: We mined all bug reports from the projects' bug tracking systems (BTSs). Mozilla, Apache, and the Linux kernel use a Bugzilla BTS, and therefore we used the provided rest API, while PHP uses a custom BTS, so we scraped its html pages. In all cases, since the amount of data is large (several GB), bulk http requests would time out, and therefore we used per-month requests.

4. Developer data: To obtain data regarding the developers involved in the respective projects, we cloned the github mirrors of the projects' repositories.

5. Social network data: To enhance our dataset w.r.t. the affiliation of reporters, we extracted relevant information from the public profiles of reporters in the LinkedIn professional networking site. More about the approach can be found in the following section.

6. Bug bounty platform data: We mined the publicly visible portion of the Hackerone[16] bug bounty platform. The relevant bounty programs for the chosen projects within Hackerone are the *Apache httpd (IBB)*[17], the *PHP (IBB)*[18] (indefinitely suspended since October 2020), and *The Internet*[19] programs (all within the scope of the *Internet Bug Bounty (IBB)* program). We found bounty information about a total of 161 CVEs in our dataset (4 for Linux, 142 for PHP and 15 for Apache), including reporter information for 1 Linux CVE and 35 PHP CVEs, for which no reporters had been retrieved from the other sources. Note that this data source does not include all bounty-related information for our study, since such information is also included in the bug reports for some projects. We present how we extracted bug bounty information from bug reports in the related results section (Section 5.5.4).

All data were collected from publicly available sources and the collection scripts as an open source project, ensuring reproducibility of our results. A graphical representation summarizing the data points collected and the links between them is provided in Figure D.1 of Appendix D.1. A summary of the number of data points collected can be found in Table 5.1.

5.4.3 *Data cleaning and pre-processing*

Data cleaning was a laborious process requiring considerable manual work. We strove to make this manual effort a one-time job by encoding the logic of the process into reusable and extendable scripts.

16 https://www.hackerone.com/
17 https://hackerone.com/ibb-apache?type=team
18 https://hackerone.com/ibb-php?type=team
19 https://hackerone.com/internet?type=team

Project	CVEs	PA	DSA	USN	RH	sf	BTS	h1	Total
Mozilla	2 195	992	218	369	1 040	1 421	209	–	2 193
Apache	249	71	27	38	12	177	—	15	197
PHP	638	—	37	31	18	393	198	142	603
Linux	2 566	201	533	562	473	1 555	—	4	1 962

Table 5.1: Breakdown of information sources. PA stands for project-specific advisories, DSA for Debian Security Advisories, USN for Ubuntu Security Notices, RH for the Red Hat Linux BTS, sf for Symantec's securityfocus.com, BTS for the project's bug tracking system, and h1 for HackerOne. The last column is the total unique CVEs that we could obtain any information for (union of all sources).

5.4.3.1 *CVEs affecting projects*

We found out that attributing CVEs to the affected projects (code-bases) is not a trivial task. Although the NVD provides information following the Common Platform Enumeration (CPE) Dictionary, some cleaning was necessary due to errors in the NVD. First, we filtered CVEs by the CPE identifier for each of the projects. For Mozilla and Apache, some CVEs reported in the vendors' security advisories were missing from the resulting set, and were subsequently added. For the Linux kernel, there were two issues that were identified and resolved:

– We noticed that the set of CVEs returned when searching with the CPE of the kernel (`linux:linux_kernel`) is missing some CVEs. On the other hand, a free text search with the keyword "Linux kernel" in CVE summaries returns quite some noise (CVEs that do not correspond to the kernel but mention it). To overcome this issue, we only added CVEs returned by the keyword search that included a reference to the git repository of the kernel. This yielded a total of 6 additional CVEs, e.g. CVE-2007-6712 that includes a CPE for `kernel:linux_kernel` (probably a mistake). The low number of additional CVEs indicates that the effect of this type of NVD errors in previous studies would be very limited.

– We noticed that some NVD entries utilizing the AND logical operator to specify vulnerable configurations erroneously labeled the operating system part of the description as vulnerable. For example, CVE-2015-0312 describes an Adobe Flash Player vulnerability that affects certain versions of the Player running on Linux (and other versions of the Player running on Windows). This is not a kernel bug, yet the json feed of the NVD wrongly labels the CPE of the operating system as "vulnerable". We filtered out 35 such instances by keeping only CVEs that refer to a kernel git repository when both another application and the kernel are included in the list of affected CPEs.

5.4.3.2 *Reporter information*

Cleaning reporter data was a multi-step process, consisting of the following:

(i.) Extracting reporter names via regular expressions, such as `.*(?= discovered an issue)` for each source (different regular expressions may be needed

for each source). At the end of this step, we have a list of strings $[s_1, ..., s_n]$, where n is the number of reporter sources (in our case the sources of Table 5.1). E.g. source$_i$: `Alice and Bob discovered an issue in the Linux kernel...` $\rightarrow s_i =$ `Alice and Bob`.

(ii.) Splitting each of the strings to pieces at the `'and'` and `','` predicates. E.g. `Alice and Bob` $\rightarrow$ [`Alice`, `Bob`].

(iii.) Keeping track of affiliations in a "smart" way using regular expressions. If an affiliation follows after several names (identifiers), then the affiliation corresponds to all of the preceding names. E.g. `Alice and Bob from Mozilla` $\rightarrow$ aff.(`Alice`) = aff.(`Bob`) = `Mozilla`.

(iv.) Keeping track of email addresses and twitter handles using regular expressions. We keep a list of all emails associated with a reporter as aliases for the same entity, since people are known to use various email addresses when filing reports or committing code [16].

(v.) Collecting reporters from all sources into a list, and removing remaining natural language phrases (that slipped through the initial matching of step (i.)), like `found by...`. Then, removing duplicates, after first stripping the strings of leading and trailing whitespaces and special characters like full stops. At this point we have a list of reporters for each CVE.

(vi.) Removing "reporters" matching generic terms (e.g. `the vendor`, `unknown`) when other reporters (not matching these terms) exist for the same CVE.

(vii.) Merging reporters with at least one overlapping email address or twitter handle. E.g. [`Alice <alice@alice.com>`, `alice@alice.com`].

(viii.) Merging "similar" reporter names, based on their Levenshtein distance, using the FuzzyWuzzy python library[20]. E.g. [`Michael Jordan`, `Michael Jordon`][21]. We merge entries over 4 characters that are 90% or more similar (based on the similarity ratio metric of FuzzyWuzzy). Manual inspection of all matches produced by this (rather conservative) rule revealed no false positives (according to our best judgment).

(ix.) Manual fixes (**This is the only step with no automation scripts**). E.g. reporter names like `Bug report XYZ` were deleted. Note that manual post-processing is most likely necessary in the general case because reporter nicknames and entries like the one above may be hard to tell apart automatically. A manual pass of the reporter list was then made to assign a label of *human*, *organization*, or *team* to each reporting entity (to the best of our judgment based on the name of the reporting entity).

20 https://github.com/seatgeek/fuzzywuzzy

21 The correct spelling is the second. Look closely at https://www.mozilla.org/en-US/security/advisories/mfsa2011-41/ to see the typo.

(x.) Mining additional affiliation information. To enhance our dataset w.r.t. the affiliation of reporters and to capture temporal changes in this field, we use information from LinkedIn. Since this step of the process involves significant manual effort to clean and validate the discovered profiles, we narrowed the scope to the top 100 reporters (with most reports) in our dataset. As these top reporters contribute significantly to the total number of reports, we judged this to be a reasonable trade-off between improving our dataset and manual effort. Attempts to fully automate the process and mine information for all reporters resulted in non-negligible noise in the returned data. For each reporter in the top 100 (reporting entities that we had not characterized as teams or organizations), we used the "people search" function provided in LinkedIn's interface, searching for each person's name followed by the key-phrase `computer security`. We then cleaned the returned results by only considering individuals working in related fields (e.g. Computer & Network Security, Computer Software, Internet) and filtering out profiles that advertise Physical Security as a skill. This brought the number of available profiles to 48. We further investigated the accuracy of the mappings by manually searching for characteristics that point to a security researcher, e.g. phrases like "developed fuzzing tools" or descriptions like "Ethical hacker". Additionally, we cross-referenced information from the profiles with information that we already possessed in our dataset regarding the affiliation of reporters, as well as information from associated presentations, blog posts, etc. that were harnessed via web searches. The result was that 40 out of the 48 profiles were classified as correct, while 8 of them as incorrect. For 7 out of the 8 incorrect cases, we were able to manually identify the correct profile, while for the remaining case we could not. At the end, we had manually verified profiles for 47 reporters that were within the top 100 most productive reporters of our dataset.

The cut-off date for the data presented in this study is the 13th of January 2021. At the end of the cleaning process, we ended up with 2 193 unique reporting entities (2 060 human reporters with the rest being 'organizations' or 'teams') associated with 4 756 CVEs. Considering that the total number of CVEs for the four projects is 5 648, our dataset has at least one reporter for ~84% of all CVEs, a figure that surpassed our initial expectations. A breakdown of the final dataset per project is provided in Table 5.2.

Project	CVEs	w/ reporter(s)	coverage
Mozilla suite	2 195	2 085	95 %
Apache httpd	249	196	79 %
PHP	638	520	81 %
Linux (kernel)	2 566	1 955	76 %

Table 5.2: Breakdown of reporter coverage

5.5 RESULTS

In the following sections, we present the analysis of the dataset that we constructed. We set off by investigating several characteristics of the distribution of reporters, both statically and over time. We then proceed to investigate indicators of reporter motivations. As stated earlier (see Section 5.2), the goal of the analysis is (a) to identify patterns and characteristics that can be used as a benchmark for the health and extent of the vulnerability hunting communities of open source projects, and (b) to provide indications on whether individual community engagement metrics can be used as indicators of health. We use raw numbers (tables), suitable plots and empirical domain-specific reasoning to assess the strength of relationships in our data. Furthermore, we use suitable statistical tests with a typical 5% significance level to assess the statistical significance of our results when applicable.

5.5.1 *Distribution*

(Concentration of reports.) We begin our analysis by investigating the distribution of the total number of reports per each reporting entity. With this investigation we explore whether the vulnerability reporting ecosystem is more centralized (suggesting more dedicated resource allocation) or decentralized (suggesting more community engagement) for the projects under study. Figure 5.1 shows

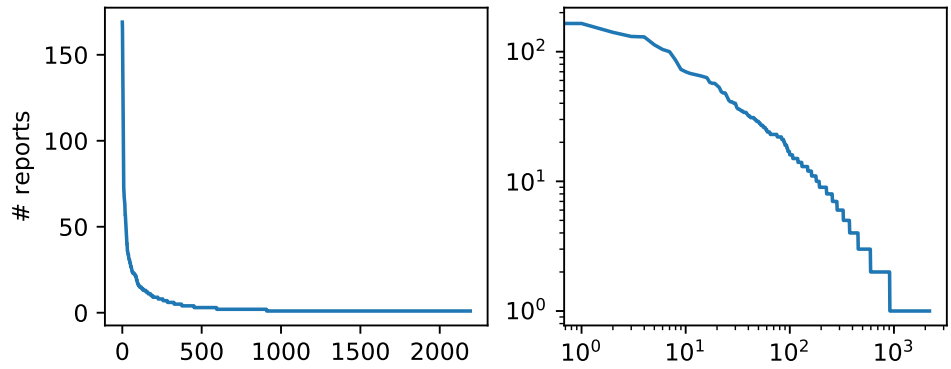


Figure 5.1: Distribution of reports per reporter in linear and double logarithmic scales (x axis: reporters ordered by number of reports).

the distribution of reports per reporter for all projects. Observing an almost straight line in double logarithmic axes in the second plot of Figure 5.1, which is an indication of a possible power-law distribution, we moved on to investigate the distribution statistically. Power laws ($p(x) \sim x^{-k}, x > x_{min}$) are heavy-tailed distributions indicative of preferential attachment behavior, which have received interest in several fields, including software engineering [60] (remember that we observed a power law distribution already in Chapter 3 when we investigated the distribution of CVEs among Debian packages). We again followed the seminal methodology for fitting heavy-tailed distributions of Clauset et al. [24] and used the Kolmogorov-Smirnov statistic to compare possible al-

ternative distributions. Considering a 95% confidence interval, we found that a truncated power law (power law with exponential cut-off with a probability density function: $p(x) \sim x^{-k}e^{-\lambda x}$) is a possible statistical fit when considering all the projects together and when considering Mozilla CVEs, while a pure power law is also a possible fit for the other projects (Linux, PHP
* and Apache) when considered individually. In all cases, as with most real-world datasets [24], other candidate distributions, like the stretched exponential ($p(x) \sim x^{\beta-1}e^{-\lambda x^{\beta}}$), cannot be statistically ruled out with the data available. This is not an issue for our purposes, as we do not focus on explaining the generative mechanism behind the distribution (which we briefly comment on in the next paragraph), but rather focus on the imbalance characteristic that heavy-tailed distributions exhibit. Detailed plots of the fits provided by candidate distributions can be found in Figure D.2 of Appendix D.

* The generative mechanism of a heavy tailed distribution such as a power law is often described by the phrase "the rich get richer" implying a form of preferential attachment. In our case this could imply that entities who have already reported many vulnerabilities are more likely to report even more[22]. These entities may be individuals or organizations with considerable means (e.g. computational capacity) or skills that would allow them to discover a large number of vulnerabilities (e.g. by discovering new classes of vulnerabilities or by developing a new fuzzer for a specific interface).

The "80/20 rule" or the Pareto principle, where 80% of the contributions (e.g. wealth, bugs, CVEs) are attributed to 20% of the population is often used as an example to portray the behavior of a power law distribution. However, power laws can be more balanced or imbalanced. The generalized principle can be described by the "$(1-p)/p$ law" [41]. In fact this ratio describes the power law distribution uniquely and provides a metric for the concentration of contributions. We empirically calculate the *CVE report concentration metric X/Y* (read as "Y% of reporters contributed X% of reports") for the projects in this study with the following results: Mozilla: 78/22, Apache: 59/41, PHP: 70/30, Linux: 72/28. We see that reports for Mozilla are more concentrated in a "core" group of reporters in comparison to the other projects. On the other side of the spectrum, reports for Apache are more balanced. We discuss how this metric can be used to describe the balance between dedicated reporters and the "many eyes" of Linus' law in Section 5.6.

5.5.2 *Temporal characteristics*

* *(CVEs and reporters over time.)* Figure 5.2 shows several temporal characteristics of the reporting process[23]. The first plot of Figure 5.2 shows the number of total CVEs (with or without reporter information) per year, for each of the

22 Here we can draw an interesting parallelism to Lotka's law [59] of scientific productivity (number of publications in a set period of time).

23 Note that to assign CVEs to years, we used the 'YEAR' portion of the CVE identifier (the YYYY of a CVE identifier that has the format CVE-YYYY-NNNNN). According to MITRE, "the YYYY portion is the year that the CVE ID was assigned OR the year the vulnerability was made public (if before the CVE ID was assigned)".

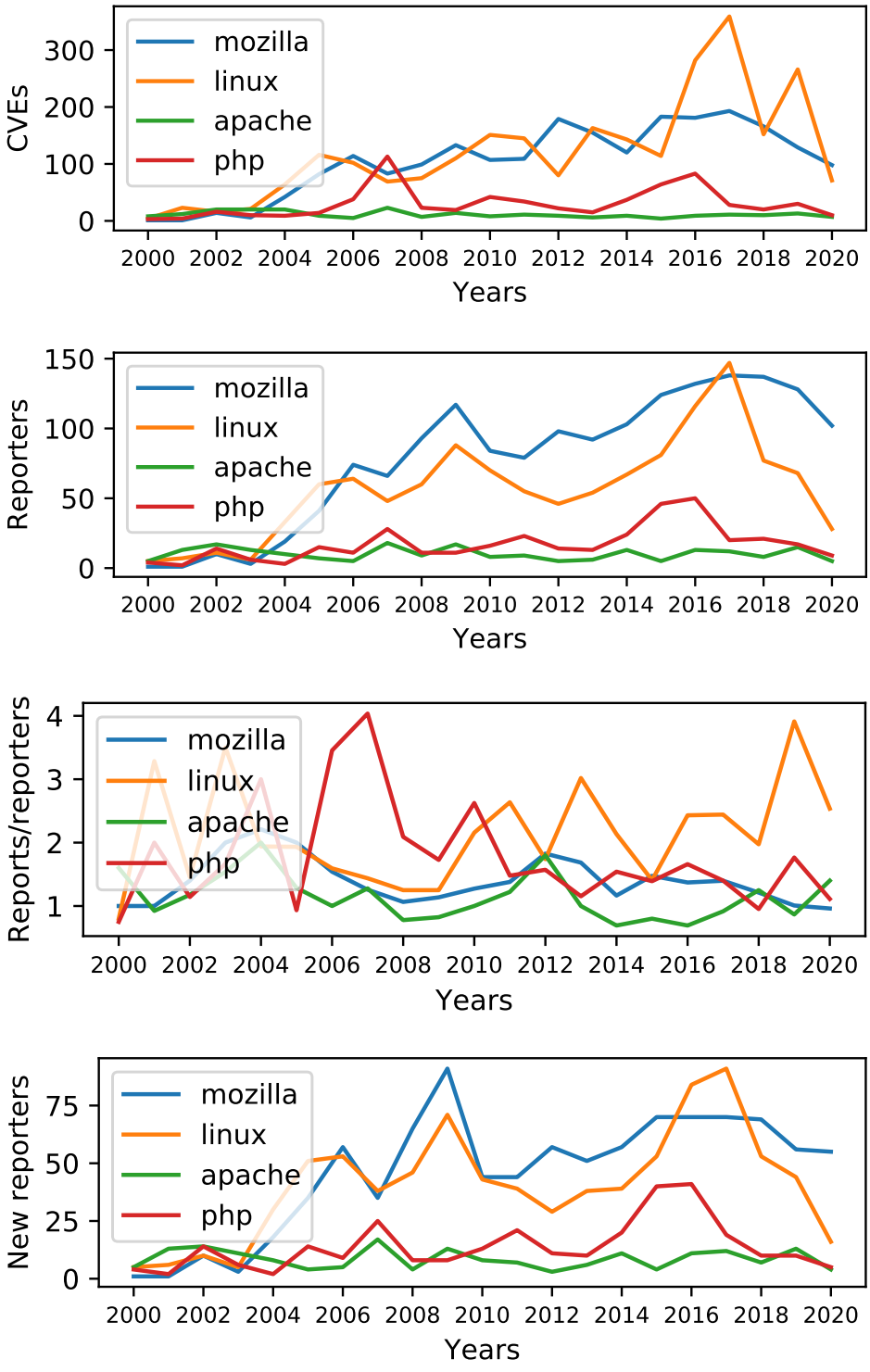


Figure 5.2: From top to bottom (all per year): # of CVEs, # of reporters, ratio of reports per reporter (# of reports divided by # of reporters), # of new reporters. Note that the drop for 2020 observed in all plots is due to the time of data collection being in early 2021 (information for some 2020 vulnerabilities may not have been published at that point).

four projects[24]. For Mozilla and Linux, we observe an overall increasing trend over time, whereas for the two other projects (Apache and PHP) we do not observe a trend, possibly due to the small number of data points available. The second plot of Figure 5.2 shows the number of reporters per year for each of the four projects. Focusing on the two projects with the most data points available (Mozilla and Linux), we can visually observe a correlation between the number of reporters and the number of reports of the first plot. Statistically, the time series of the first two plots are strongly correlated for all four projects (Spearman's Rank Correlation coefficient $\rho > 0.8, p < 10^{-5}$). This observation agrees with our expectation that human effort (here: indirectly expressed by the number of reporters) may be the dominant factor affecting the number of discovered vulnerabilities in large projects. However, it is clear that we cannot claim anything regarding causation between the two variables purely based on the data available. An alternative interpretation of the observed correlation would be that vulnerabilities are getting easier to find (e.g. due to an increase of software complexity, discovery of new vulnerability classes), and therefore more people are reporting them (while the total effort, e.g. expressed by the number of people looking for vulnerabilities, remains constant). Most likely, a combination of several factors is at play here, with human effort having a significant effect.

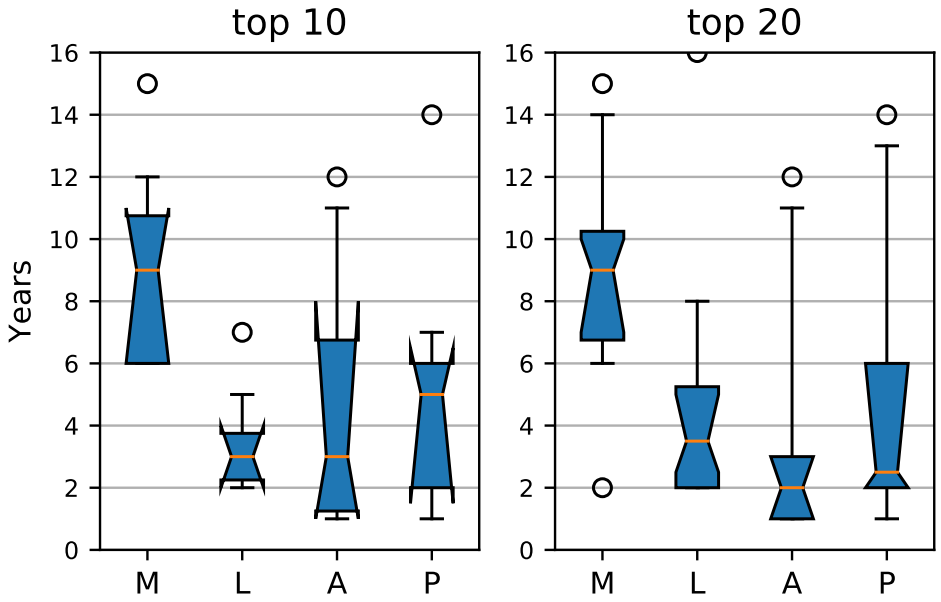
* The third plot of Figure 5.2 shows the ratio of reports per reporter over time (total number of reports divided by total number of reporters per year). The intention behind this metric is to capture a notion of community engagement, in the sense of: "on average X people needed to work to find each vulnerability". We discuss whether such a metric can be used as an indicator of quality in Section 5.6 (short answer: no – at least by itself). As expected (due to the correlation noted above), we observe no noteworthy trend in the plot, and the value varies between just under 1 and 4 reports for all projects, with the Linux kernel having a slightly higher ratio during the last years. Note that a CVE can have multiple reporters, thus this number is not bounded by 1. The fourth plot shows an interesting phenomenon: in particular for the Linux kernel and Mozilla, there is a constant influx of new reporters ranging from 40 to 90 per year (roughly half of all reporters for each calendar year are first-time reporters). Thus, new reporters are significant contributors to the process.

(Period of engagement.) Since, as we showed, the distribution of reports per reporter is heavy-tailed (most reporters have reported only a few bugs and few reporters have reported most bugs), for this part of our investigation we focus on this heavy tail by looking into the *top* (with most number of reports overall) reporters for each project. Figure 5.3 includes 2 box plots.

* The first plot captures the period of engagement (years between first report and latest report, plus one) of the top 10 reporters for each of the four projects. We observe that top Mozilla reporters have a significantly[25] longer period of

24 Since we collected our data in early 2021, the analysis for some 2020 entries had not been published yet in the NVD, explaining the lower number of entries for that year compared to previous ones.

25 Note non-overlapping 95% confidence intervals for the median in Figure 5.3.



* **Figure 5.3:** Period/duration of engagement for the top 10 and top 20 (human) reporters (time in years between their first and more recent report until now – increased by one), for each project ([5,95] whiskers). Letters in the x axis are the initials of the corresponding projects (Mozilla, Linux, Apache, PHP). The 95% confidence intervals for the median, calculated via bootstrapping (10 000 times), are marked with notches.

engagement compared to top Linux reporters. The median for Mozilla is more than 9 years, whereas for Linux only 3. We can make the same observation when considering the top 20 reporters for these two projects (second plot). In this case Mozilla has a significantly longer duration of engagement compared to the other three projects. Note that a reporter's last report stands for their last report until now; we can not know if they make new reports in the future. Therefore, these plots provide a lower bound on the duration of engagement. Overall, we see that Mozilla has a more 'long-serving' (stable) base of long-term regular contributors of vulnerabilities, while for the other projects, even their top contributors report in bursts and stay engaged for a shorter amount of time.

5.5.3 *Specialization*

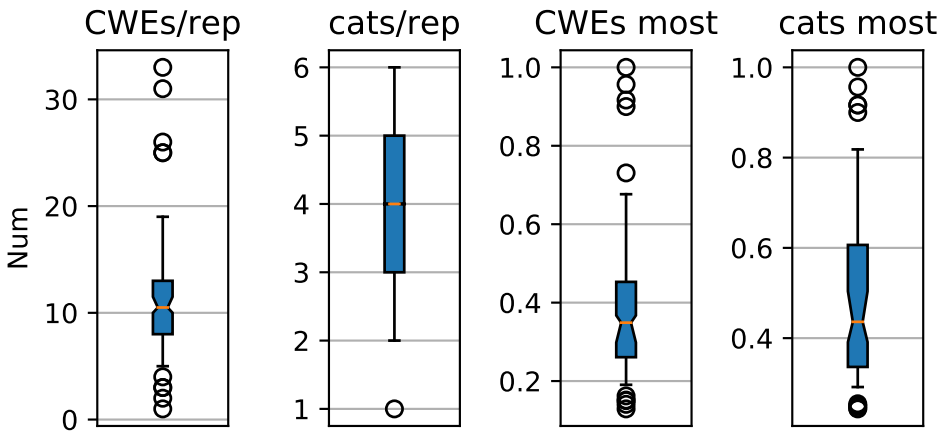
(Project specialization.) Only 83 reporters (out of a total of more than 2 000) have reported a vulnerability for two or more of the projects. This number goes down to 14 entities who reported vulnerabilities for three or more of the projects, and only one entity (iDefense, which was a bug-bounty program, so probably includes multiple entities) reported a vulnerability for all four of the projects under investigation. Thus, reporters to multiple of these four FLOSS projects are rare, i.e. reporters are generally specialized w.r.t. the project they are testing.

(Type specialization.) To investigate reporter specialization w.r.t. CVE types, we extract the Common Weakness Enumeration (CWE) number of each CVE from

the information available at the NVD. Since some CWE types are closely related or have changed over time, and since we are interested in a more high-level classification of vulnerability types, we follow the approach of Chapter 4 to map each CWE to one of the 6 following high-level categories:

1. Memory and Resource Management (e.g. CWE-119: "Improper Restriction of Operations within the Bounds of a Memory Buffer")
2. Input Validation and Sanitization (e.g. CWE-20: "Improper Input Validation")
3. Code Development Quality (e.g. CWE-369: "Divide By Zero")
4. Security Measures (e.g. category CWE-310: "Cryptographic Issues")
5. Concurrency (e.g. CWE-362: "Race Condition")
6. Other

We provide an overview of the number of different CWE types and categories found by reporters with 20 or more reports (total of 90 reporters related to 1 974 CVEs) in the box plots in Figure 5.4. Half of all these reporters have reported issues of 10 or more different CWE types, or alternatively 4 or more categories. Half of the reporters have a specific CWE type that accounts for more than 30% of their reports, or alternatively they have a specific higher-level category that accounts for 45% or more of their reports.



* **Figure 5.4:** Number of different CWEs and categories (cats) for each reporter. Third column: for each reporter, what portion of their reports falls into the CWE/category with the most reports (dominant).

To investigate whether the distribution of reporters per category varies between reporters, we compared the distribution of categories of each individual reporter with at least 20 reports, against the overall distribution of categories for the project that the reports concern. We only looked into reporters with at least 20 reports in order to be able to make statistical arguments about the distributions, and we looked for reports in each project individually to account for

different categories being more common in different projects (e.g. the number of concurrency bugs is negligible in projects other than the Linux kernel). We employed a Chi-squared test[26] to assess whether the distribution of categories corresponding to reports from a given reporter deviates significantly ($p<0.05$) from the expected (taken for the project as a whole). Then, for those reporters whose distribution deviates, we checked if more than half of their reports fall into the same category (a simple measure of strength of the significant deviation observed), signaling a specific focus. The results are summarized in Table 5.3.

Project	reporters > 20	# deviate	# focused
Mozilla suite	49	33	21
Apache httpd	0	0	0
PHP	4	1	1
Linux (kernel)	18	16	13

Table 5.3: Deviations from expected categories. From left to right: reporters with 20 or more reports; out of them, reporters with a deviation w.r.t. categories; out of the latter, reporters with a specific focus category. Note that the values in the second column are a subset of the values of the first, and the values in the third column are a subset of the values in the second.

Due to the low number of observations for Apache and PHP, we focus on Mozilla and Linux. For Mozilla, out of the 21 reporters who exhibited a particular focus, 13 focused on memory-related issues (category 1), 3 focused on issues related to security measures (category 4), while 5 focused on "Other" issues. For Linux, out of the 13 reporters with a significant focus, 3 focused on memory-related issues while 10 focused on issues related to security measures. Overall, results indicate that a significant portion of the most productive reporters are specialized in a specific category of vulnerabilities and that there are two main categories of specialization: *memory* and *security measures*. We can conjecture that reporters specialized in memory issues are the ones who develop and operate several fuzzing mechanisms that have become popular during the last years, while reporters specialized in security measures are the ones looking specifically for issues that do not usually cause crashes (and therefore cannot be detected by traditional fuzzing tools); rather these issues have to do with permissions, cryptographic implementations, data leakage, etc.

* Note that in this section we focused on reporters with 20 or more reports, corresponding to 90 out of a total of 2 193 reporters, in order to be able to make statistical arguments regarding reporter specialization w.r.t. vulnerability type. It would be interesting to study whether reporters with fewer reports are also specialized, however this is not possible with our dataset, and would likely require interaction with the reporters.

26 Chi-squared being a non-parametric test, we do not need to make any assumptions about the distribution of the data.

5.5.4 *Motivations*

* *(Bug bounty programs.)* Apart from bug bounty information collected from the HackerOne platform (see Section 5.4) we further investigate the projects' bug tracking systems for relevant information. Mozilla, in particular uses a '*sec-bounty*' flag to mark bugs that were considered for a bounty in Mozilla's bug bounty program. We only consider bugs where the '*status*' property of this flag is set to '+', meaning the bounty was awarded[27]. We collected such bugs and matched them to their associated CVEs in our dataset (where applicable). For the other projects that either fully (Apache) or partly (Linux) use a Bugzilla instance to track bugs, we did not find any bugs that were awarded bug bounties, following the same approach. For PHP, Apache, and Linux we also ran a regular expression search (string includes 'bounty') in the '*comments*' section of their bug reports but found no additional information[28].

Table 5.4 shows the number of CVEs associated with a bounty for each project. Because we want to gain insights into the incentives of reporters, we provide:

- a low estimate, which corresponds to the confirmed number of CVEs for which a bounty was awarded
- a high estimate, which corresponds to the number of CVEs by a reporter who was awarded a bounty for at least one of their reports (at any time)

Project	CVEs	bounties low-high	% low-high
Mozilla suite	2 195	589–1 206	27–55
Apache httpd	249	15-16	6-6
PHP	638	142-208	22-33
Linux (kernel)	2 566	4-50	0-2

Table 5.4: Reports with bounties per project. Includes a low estimate of confirmed bounties given and a high estimate of reports by a reporter who has received a bounty at least once.

We can observe that the effects of bug bounties vary greatly between the projects, with Mozilla, being one of the pioneers of bug bounties in the FLOSS community, benefiting greatly from the program. However, in each of these projects, bounties were not given to more than half of the reporters. Furthermore, the Linux kernel does not depend on bounties for vulnerability discoveries at all and still maintains a consistent influx of new reports and new reporters. Thus, while bug bounty programs are helpful, reporters have additional motivations.

27 Personal correspondence with the Mozilla security team confirmed that this approach is valid to track bounties from mid-2010 onward, however a gap between 2005 and 2010 exists, for which no data is (and can be) provided.

28 Interestingly, for Linux there were some results returned by the search, however these were for bug reports that mentioned the existence (or need) of a bounty to fix the issue (see e.g. https://bugzilla.kernel.org/show_bug.cgi?id=195303#c98).

* These motivations may include producing academic research, gaining reputation in the security community or volunteering for the health of FLOSS ecosystems. Finally, they may be employed by the organizations that are either behind the development of the projects or have an immediate interest in their security. We investigate the latter point in the next paragraphs.

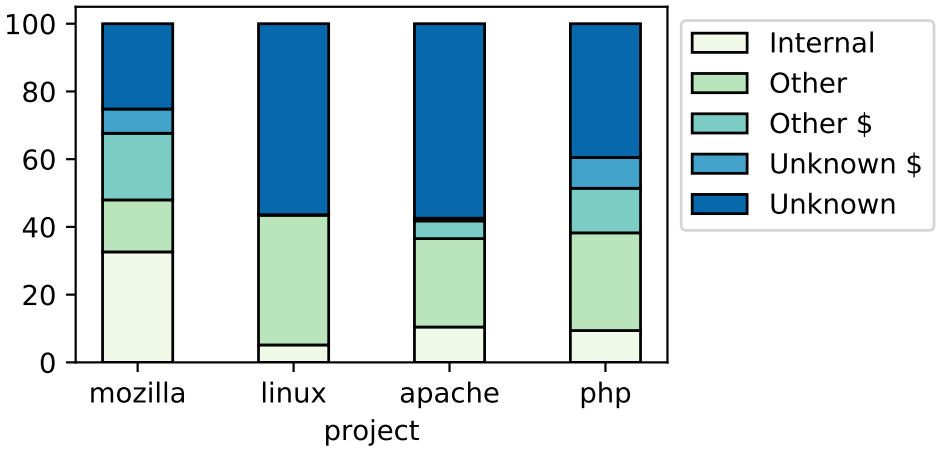


Figure 5.5: Affiliations associated to CVEs. Internal is for reporters affiliated to the organization behind the project, Other is for reporters affiliated to other organizations, and Unknown is for CVEs which could not be associated to an affiliation. We mark with $ the subsets of the two latter categories that were awarded bug bounties.

(Affiliations.) We linked reporters to affiliations based on (a) information collected from the sources of Section 5.4.2, (b) additional information from reporters' online professional profiles – when applicable (as described in Section 5.4.3). A vulnerability is internal iff (a) the reporter *matches* the organization (e.g. "Mozilla" or "Mozilla Corporation", etc. for the Mozilla suite), or (b) at least one of its reporters has an affiliation matching the organization (when temporal data for affiliations is available, e.g. via information from an online profile, then an additional constraint for the time of employment is applied).

Looking at Figure 5.5, we observe differences between the projects. A significant portion of Mozilla CVEs originated from within the organization (33%), while this portion is much smaller for the other projects (<10%). The most significant external contributors to Mozilla reports are Tencent and Google. The Linux kernel has the highest percentage of reports from reporters with known affiliations that did not receive bug bounties. A number of companies/organizations contribute significantly to the security of the Linux kernel, with the most notable being Google, Qihoo 360, and Red Hat. Also for the kernel, a comparably large number of CVEs could not be associated with an affiliation, potentially pointing to a higher engagement of hackers working on a volunteer basis.

(Commits and bug reports.) We found at least one commit in any of the projects for 730 out of a total of 2060 human reporters in our dataset (35%). 559 reporters have made 10 or more commits (27%), while 382 (19%) have made 100

or more commits. While reporters are not regular committers in their majority, a significant percentage is actively contributing to the codebase.

One hypothesis we could make is that most vulnerability reporters find and report other kinds of bugs (non-CVE) as by-products of the process, and that * these bugs are in some way different than other non-security bugs. If this hypothesis is true, then related metrics could be used to measure vulnerability-hunting effort, in the sense that even if no (or few) vulnerabilities (CVEs) are reported, the reporting of such non-security bugs would indicate security-related efforts. In the following, we investigate this hypothesis.

Project	Reporters	Bug Reporters (%)	Median
All	2060	810 (39%)	5
Mozilla suite	917	394 (43%)	8
Apache httpd	160	34 (21%)	4
PHP	277	65 (23%)	3
Linux (kernel)	790	151 (19%)	2

Table 5.5: Percentage of vulnerability reporters (excl. reporters marked as 'teams' or 'organizations') with non-cve bug reports and median of such bugs per reporter.

A significant portion (39%) of vulnerability reporters created at least one non-CVE bug report in one of the projects' bug tracking platforms. Between the projects, percentages vary between 19% and 43% (Table 5.5). For the Mozilla Suite, this value is clearly higher with 43% (19-23% for the other projects). Mozilla reporters also report more non-CVE bugs than others (median for Mozilla is 8 compared to 2-4 for the other projects).

Project	Severe		Resolved		Keywords	
	reps	rest	reps	rest	reps	rest
Mozilla suite	**20%**	13%	**40%**	34%	**23%**	11%
Apache httpd	12%	**20%**	**47%**	32%	**15%**	10%
PHP	-	-	**46%**	11%	**22%**	12%
Linux (kernel)	11%	**16%**	**31%**	27%	**21%**	15%

Table 5.6: Differences between bugs by vulnerability reporters (reps) and others (rest) as a percentage of the total for each class. All statistically significant ($p < 0.05$ for the Chi-squared test). Severe are bugs with *critical*, *major* or *high* severity (except for PHP for which no severity field exists). Resolved are bugs marked *fixed* in bug reports (except for PHP for which no such field exists, and therefore we considered bugs with associated fixing commits).

To analyze if bug reports by vulnerability reporters differ from bug reports by others, we used the fields *severity* and *resolution* of bug reports (assuming that vulnerability reporters report more severe bugs that are also more likely to be resolved). Additionally we searched for the keywords *memory*, *crash* and

security in bug descriptions, assuming that these words may occur more often for bugs discovered during security testing. The results are summarized in Table 5.6. We observe a higher percentage of severe bugs only for Mozilla, with the tendency reversed for Apache and Linux. Regarding *resolution*, for each of the projects, a higher percentage of bug reports created by reporters are fixed. For keywords, reporters of all projects have created a (slightly – but statistically significant) higher percentage of bugs mentioning *memory*, *crash* or *security*. Overall, although we identified statistically significant differences (see Table 5.6) between bugs created by vulnerability reporters compared to bugs created by others, the magnitude of the differences is small. Additionally, as stated earlier, only a minority of vulnerability reporters also reported other bugs, with the percentage varying significantly between projects. As a conclusion based on the two previous statements, we cannot support the hypothesis that by-products of vulnerability hunting in a project's bug tracking platform can be identified and utilized, in order to assess the expended vulnerability hunting effort.

5.5.5 *Summary of main findings*

We conclude this section with a compact summary of our main findings in this chapter.

Summary of main findings

The distribution of reporter contributions can be described by a power law, meaning there are a few reporters responsible for most reports, while most reporters report only a few vulnerabilities. The number of reports is correlated with the number of reporters over time, while first time reporters account for a significant portion of the reports on a yearly basis. Regarding the period of engagement, for Mozilla, top reporters stay involved for a median of more than 8 years, significantly more compared to the other 3 projects. Also, reporters are specialized w.r.t. the type of a vulnerability. Regarding motivations, bug bounty programs contribute significantly, yet bounty-related reports are a minority in all projects. Furthermore, a minority of reporters are affiliated to the organization behind the projects, while 35% of reporters have also committed to the project's repository, and 39% of reporters created a non-CVE bug in the project's bug tracking platform.

5.6 IMPLICATIONS AND DISCUSSION

In this section, we discuss some implications of our results.

Observations as a benchmark and recommendations for new projects. We saw that the FLOSS projects in our study depend both on a dedicated set of "core" reporters with many reports, as well as on one-off contributions from a large set of reporters ("many eyeballs"). Furthermore, FLOSS projects depend

on attracting a steady influx of new reporters into their communities. Therefore, projects should make sure that they continuously attract both types of reporters, e.g. via having dedicated security resources and planning in addition to engaging with volunteer hackers. Depending on the type of the project, the motivations of reporters may vary (as we observed for the projects in this study), and as a result the best ways to attract them may also vary. If the project is used as a core part of the operation of other organizations (e.g. the Linux kernel), then these organizations may contribute to creating the dedicated "core" of reporters. Otherwise, more investment of own resources or bug bounty programs will be needed (e.g. as in the case of Mozilla). Also, productive reporters seem to be specialized w.r.t. the types of vulnerabilities they are looking for (especially regarding memory-related issues and security issues).

*

> Overall, our findings indicate that an effective strategy for FLOSS projects is to continuously attract a diverse set of vulnerability reporters, both dedicated researchers and opportunistic community members.

Community engagement metrics as indicators of quality. An ideal numerical "vulnerability-hunting effort" metric would provide the community with a powerful tool in order to measure the security of software projects. The complexities and per project peculiarities that we showcased in this chapter suggest that a singular such metric may be unattainable (more details as to why follow in later paragraphs). However, some of the metrics we investigated in this chapter – in combination with qualitative characteristics discussed in the previous paragraph – provide useful indications regarding the quality and intensity of the vulnerability-hunting process for a project. Specifically, the $(1-p)/p$ ratio of the distribution of the number of reports per reporter (generalized Pareto principle) can be used as a metric for the concentration of contributions in a project. We saw that all projects in our investigation showed some imbalance w.r.t how many reports are contributed by each reporter. There are many factors (familiarity with a project, use of automated tools, discovery of vulnerability patterns) that could support a preferential attachment mechanism that creates a heavy-tailed distribution of reports per reporter. A very high concentration of reports (e.g. 90/10) would indicate low participation of the "many eyeballs" of a community to the vulnerability reporting process. This may or may not be a bad sign for the health of the security ecosystem of a project, depending on observations that can be made regarding the motivations of reporters. Such observations can be made following an analysis like the one we introduced in Section 5.5.4. A very low concentration (e.g. 50/50) would indicate a lack of dedicated resource allocation (or lacking effectiveness of those resources).

Our results suggest that metrics such as the popularity of a project, or the number of developers, do not directly relate to the number of vulnerability reporters, since most reporters were not otherwise involved in those projects. Anecdotal evidence, such as the Linux foundation's executive director's Jim Zemlin commentary on the Heartbleed vulnerability of OpenSSL: "In these

cases the eyeballs weren't really looking," seem to support this suggestion. Merely being a popular project does not automatically translate to having an active and adequate community of vulnerability reporters. This would also mean that the use of the popularity of a project (e.g. install base) as a metric for vulnerability-hunting effort (as in some effort-based vulnerability discovery models) may lead to uninformative or even misleading results. A detailed analysis, as the one presented in this chapter requires human involvement for the interpretation of its results[29], yet we believe it provides more actionable and valuable indications regarding the health of the vulnerability-hunting community of a project. Thus, we believe that the methods presented in this chapter can help construct a "general profile" of the health of a project's vulnerability reporting ecosystem. Such a profile can be readily included to enhance the quality of studies routinely published by security companies that try to assess the relative security of software products. Such "studies" usually miss the mark [54] (up to now) and provide one-dimensional analyses, most commonly counting the number of reported vulnerabilities.

*

> Profiling the reporting ecosystem of a project, as introduced in this chapter, can find immediate practical impact in providing indications regarding the health of the security ecosystem of software projects. Specifically, the $(1-p)/p$ ratio can be used as a metric for the concentration of contributions in a project, and as a consequence, act as an indicator of (security) quality.

* **Building theories.** One of the motivations behind this study was the hypothesis (supported by anecdotal evidence and prior results – see also Chapter 3) that human effort expended in the vulnerability discovery process is the primary factor influencing the vulnerability discovery rate. An interpretation of this hypothesis in simplistic terms (not considering differences in individual skills, amount of time invested, available hardware, etc.) would be that given a certain software project and a specified amount of time in the future, the number of vulnerabilities discovered will be proportional to the number of people looking for them (or maybe better the number of "skill-normalized work hours" invested). Generalizing this interpretation, consider the following example.

* *(A notion of quality.)* Let us consider two different projects with the same number of discovered vulnerabilities in a specific time period in the past. We could try to compare these projects based on a notion of "quality of security" defined as the average difficulty of finding a new vulnerability. Then we could state that the average difficulty of finding a vulnerability in that time period – and as an extension the "average quality" of the software during this time period – would be proportional to the number of people looking for vulnerabilities for this software during this time-period. Now, even when the number of vulnerabilities discovered for each of the projects may differ, the same line of thought of pro-

29 A certain degree of automation may be achieved following, e.g. a rule-based approach, however this is outside the scope of our work.

portionality would allow us to state that the "quality" of each software project would be proportional to the average ratio of people required to discover a vulnerability. For example, assume project X had 100 vulnerabilities discovered during the last year and another project Y had 50. Also, assume we know that 5 000 people searched full-time for vulnerabilities in project X during the last year while 1 250 in project Y. Calculating, this would mean that on average 50 people would need to work full-time to find a vulnerability for project X in comparison to 25 for project Y. Consequentially, we could state that project X is (or at least was on average during the last year) "twice as secure" as project Y based on an "average difficulty of discovery" notion of security.

* Although simplistic, a metric based on the line of thought presented above would express a notion of quality. Of course, the main obstacle towards computing such a metric is that – especially for FLOSS projects – we can hardly have any idea about how many work-hours were invested overall in the vulnerability finding process during a specific time period. All the information that is feasibly available (and can be collected as presented in this chapter) is how many entities (people or sets of people) actually succeeded in discovering (and reporting) vulnerabilities. The question is: can the readily available ratio of reports per reporter (total number of reports divided by total number of reporters in a specific amount of time, e.g. per year as shown in Figure 5.2) be used to estimate the average "difficulty" of finding a vulnerability?

* The short answer is *not quite*. The heavy-tailed characteristics of the distribution of reports per reporter (Section 5.5.1 – recall the $(1-p)/p$ ratio also discussed above) indicate that there can be a large variance in the productivity of reporting entities. One explanation based on our observations is that some of the reporting entities are large teams of people (e.g. Google, Mozilla) while others are individuals. Consider the following comparison of the Linux and Mozilla security ecosystems to better understand the issues that arise.

* *(Linux vs. Mozilla.)* Let us try to compare Linux and Mozilla in such a manner. In Figure 5.2, we see that the two projects have a similar number of reported CVEs during the last years, while Mozilla has a higher number of reporting entities for each year (and consequently a higher number of reporting entities per CVE). Does this evidence suffice for us to conclude that it was more difficult to discover a vulnerability in Mozilla compared to Linux based on this metric – and therefore Mozilla is in a way more secure? Not quite. As we saw in Section 5.5.4, a significant number of Linux reporters were big organizations (or individuals affiliated to big organizations). Also, a significantly higher number of Mozilla vulnerabilities were discovered "in house" (33% compared to <10%). Overall, these observations suggest that the effort expended in finding vulnerabilities for Linux is comparatively higher per reporting entity than for Mozilla. Overall, we can say that neither of the two projects is strictly more secure, or alternatively, the two projects are in the same class when it comes to security (again defined as per the line of thought above).

*

> Uni-dimensional metrics of effort fail to capture the complexity of real-world security. However, in this chapter we showed that a combined assessment of several parameters (mainly but not limited to: the $(1-p)/p$ ratio, # of CVEs normalized per reporter, reporter affiliations, ratio of internal/external reporters)—in particular in a comparative assessment between different projects—can provide valuable indicators regarding the health of a project's security ecosystem (w.r.t. vulnerability finding).

* **Additional comments on the Linux kernel.** At the end, we want to return to the peculiarities of the Linux project. We reported in Section 5.5.4 that Linux is the only project in our study with a negligible amount of reports associated to bug bounties. However, some Linux kernel vulnerabilities may fall within the bounds of Google's Android Security Rewards Program[30], which was launched in 2015 to reward reported vulnerabilities affecting Android devices. To the best of our knowledge, no publicly available information exists regarding the specific CVEs, reporters, and bounty amounts rewarded. We can consider, as an upper bound for the number of Linux kernel CVEs that may have been awarded a bounty through this program, the number of CVEs with reporters who were acknowledged in Android Security Bulletins (we already referred to this source in Section 5.4). Such an acknowledgement exists for 201 out of a total of 1 962 Linux CVEs (~10%) in our dataset.
* Given the paramount importance of Linux as the underlying operating system (kernel) of many digital infrastructures, one may wonder whether the existence of a dedicated bug bounty program would help improve security. Indeed, Google announced in mid-2020 that they expanded their rewards program to cover open source dependencies of their products (incl. Linux)[31], also providing an update in late-2021 announcing new bounties for the Linux kernel specifically (of up to 50 000 USD)[32]. It would be interesting to observe how these recent developments will affect vulnerability reporting for the Linux kernel in the following years. A dedicated study on the impact of such initiatives, as well as new tools, on the security of the Linux kernel, would be interesting future work.

5.7 THREATS TO VALIDITY

Below we systematically document possible threats for the validity of our results.

30 https://bughunters.google.com/about/rules/6171833274204160
31 https://security.googleblog.com/2020/05/expanding-our-work-with-open-source.html
32 https://security.googleblog.com/2021/11/trick-treat-paying-leets-and-sweets-for.html

5.7.1 *Threats to construct validity*

Reporters vs. discoverers: For the discussion of the implication of our results, we assumed that the people/organizations who get credited with reporting a vulnerability are the ones that discovered it. We believe this to be a valid assumption macroscopically, although we note that it may not be universally true.

5.7.2 *Threats to internal validity*

Best-effort: We collected our data from multiple sources and underwent a rigorous cleaning process. Understandably, this process was best-effort and we welcome corrections (and additions) by the community on our publicly available dataset.

Inherent noise and incomplete data: Our information sources (NVD, security advisories, bug reports, etc.) are manually curated and therefore subject to errors and omissions. Furthermore, we focused our investigation on vulnerabilities that received a CVE identifier. Although the CVE database is supposed to "fully cover" the projects in our study, it is known that a significant amount of vulnerabilities do not get a CVE. Although we do not expect these vulnerabilities to differ significantly (w.r.t. the characteristics we studied) to the ones in our study, future efforts can be driven towards expanding our dataset to include more vulnerabilities. Furthermore, other sources for information on security reporters, bug bounties, or affiliations may exist that we did not consider in our study. A platform to systematically gather and continuously update such information would be a worthwhile goal for the community (e.g. as an extension of the NVD).

5.7.3 *Threats to external validity – Generalization*

We investigated four popular FLOSS projects. In particular, these are among the largest FLOSS projects with longevity, so these results may not apply to smaller, newer FLOSS projects. We saw various differences in the reporter characteristics for each project, and therefore additional studies are required to have a more general understanding of the FLOSS security ecosystem. However, our methodology is largely automated and should be applicable to any other FLOSS projects that keep accurate records of their vulnerability reporters, which can significantly reduce the overhead of future studies.

5.7.4 *Threats to reliability*

* The manual assignment of human/organization/team labels (step ix. in Section 5.4.3.2) and the manual verification of LinkedIn profiles for the 48 profiles returned by the search (step x. of Section 5.4.3.2) were performed by a single person. However, we consider the impact of divergence in those aspects to be

small and to have no effect on the reliability of the results presented in this chapter. We make our code and dataset publicly available to enable reproducibility of our results.

5.8 CONCLUSION

In this chapter, we performed the first (to the best of our knowledge) large-scale study on FLOSS vulnerability reporters, going beyond the closed communities of bug-bounty programs. We investigated several aspects regarding the trends, backgrounds, motivations, and behaviors of vulnerability reporters. We identified qualitative characteristics that can act as benchmarks for healthy vulnerability-finding ecosystems. We also identified a quantitative metric that can provide indications regarding the health of the vulnerability-hunting ecosystem of a project, although further studies with even bigger datasets are required. As an independent contribution, we demonstrated that characteristics of vulnerability reporters can be studied through information mined from several publicly available online sources. Our approach is mostly automated and our data and scripts are open to the public, so future researchers or FLOSS project coordinators can further build upon this study.

In conclusion, our results suggest that individual metrics for vulnerability-hunting effort without context will fail to capture the unique characteristics of the process. On the other hand, comprehensive case studies, shedding light on multiple potentially co-dependent aspects of the process, can provide useful insights. The methodology we presented in this chapter provides a blueprint for
* such approaches. In the following table (Table 5.7) we summarize the key quantities identified during the empirical study presented in this chapter, along with comments on the information they convey in relation to the health of a project's vulnerability finding ecosystem. More research is required to better understand the relation of these quantities with each other and how they can systematically be used to assess and help improve the security of FLOSS ecosystems.

Parameter	Comments
1. $(1-p)/p$ ratio (5.5.1)	• Provides a measure for the concentration of reports among reporters for a given project.
2. Correlation over time # CVEs ~# reporters (5.5.2)	• Can be used as an indication that an increase in the number of reports is (at least partly) attributed to an increase in the expended effort.

3. $\frac{\#\text{ CVEs}}{\#\text{ reporters}}$ per year (5.5.2)	• Can be used as an indication (together with supplementary information regarding, e.g. the affiliations of reporters – as discussed in Section 5.6) regarding the difficulty of finding vulnerabilities in a given project.
4. $\frac{\#\text{ first-time reporters}}{\#\text{ reporters}}$ per year (5.5.2)	• Provides a measure for the magnitude of the effect of new reporters coming into the ecosystem.
5. Duration of engagement for top reporters (5.5.2)	• Can indicate whether the relationship of reporters to the project is long-term or comes in bursts.
6. $\frac{\#\text{ specialized reporters}}{\#\text{ reporters}}$ (5.5.3)	• Can provide indications regarding areas where more specialized personnel is needed
7. $\frac{\#\text{ CVEs w/ bounty}}{\#\text{ CVEs}}$ (5.5.4)	• Provides a measure of the impact and effectiveness of employed bug bounty programs
8. $\frac{\#\text{ CVEs w/ internal reporter}}{\#\text{ CVEs with external reporter}}$ (5.5.4)	• Provides a measure for the quantity and effectiveness of the resources dedicated by the vendor in comparison to resources attracted via other means.

* **Table 5.7:** Summary of notable parameters identified in this chapter. The section where the relevant results are presented is given in brackets.

*

Key takeaways

In this chapter, we showed how to analyze the vulnerability reporting ecosystem of FLOSS projects using publicly available data. Such analysis can provide indications regarding the health of a project's vulnerability finding ecosystem. In particular, in this chapter we showed that the distribution of reports among reporters ($(1-p)/p$ ratio) directly provides information regarding the concentration of contributors, which when considered in combination with other factors (reporter affiliations, ratio of

internal/external reports, reports associated with bug bounties – see also discussion in Section 5.6) can provide *indications* regarding a project's quality of security.

6

CONCLUSION

Having means to objectively measure the *security quality* of software would help improve software security in many ways. Specifically, being able to compare security properties of software programs would not only enable users to make more informed decisions according to their needs, but—most fundamentally—would also enable vendors to "charge" for security, generating incentives for better security and leading to significantly increased efforts. Furthermore, better ways to measure software security, including its development over time, would enable us to assess the effectiveness and impact of employed tools and practices. For example, our results regarding the non-depletion of vulnerabilities during the lifetimes of stable releases (the *Tip of the Iceberg* effect observed in Chapter 3) suggest that more attention into 'security by design' efforts is needed (e.g. memory-safe programming languages). Such efforts would avert the *introduction* of vulnerabilities in the code and make them rarer. Then, the impact of ever-improving automated testing tools would reach its full potential, as more time and resources would be available to properly examine and address the outputs of such tools.

In this dissertation we presented new approaches towards extracting informative metrics regarding the security quality of software. Although measuring security is very hard (security being complex and multidimensional) our approaches mark a small—but significant—step forward towards achieving progress. Overall, there is still a lot to be done in the area. Real-world security of deployed systems does not only depend on the difficulty of finding new vulnerabilities in the software they use. Rather, it involves numerous considerations, e.g. regarding the usability of the software, the speed and quality of the patch development and deployment processes, the training and competence of the users and administrators, etc. Therefore, further progress towards better security metrics and measurement methods requires not only the improvement and further development of the approaches introduced in this dissertation (from a 'vulnerability-finding' perspective), but also the development of approaches that capture other considerations (some of which we mentioned above).

6.1 SUMMARY OF CONTRIBUTIONS

In this dissertation we provided three new approaches to measure relevant aspects of software security:

- In Chapter 3 we provided a method for creating a dataset of vulnerabilities affecting stable releases of the Debian GNU/Linux distribution of software. We used this dataset to perform a large-scale empirical study on vulnerability discovery rates over time, trying to answer fundamental questions regarding the longitudinal development of security in popular FLOSS projects. Specifi-

cally, the research question we tried to answer in this chapter was: *Are stable releases maturing over time?* We observed no definitive decreasing trend in the vulnerability rate of selected packages (php5 and openjdk-7) and a distribution as a whole (Debian 7 'Wheezy') during the duration of a stable release – on the contrary, for most of the cases we identified an increasing trend. We also did not observe an increasing trend in the bug bounties offered for FLOSS projects in a popular bug bounty platform.

Our results suggest that software is not maturing fast enough for it to be observable during a given release (~3-5 years) yet. We termed this effect of a seemingly infinite amount of vulnerabilities during a practical time-frame after the release of a software, the *'Tip of the Iceberg'* effect. Further work is needed to investigate whether the effect can be observed for other software packages or whole distributions and what conditions may indicate its existence. Our findings indicate that metrics and models based on the rate of vulnerability discoveries are more likely to be influenced by external factors (e.g. vulnerability finding effort) than be good indicators of quality. Overall, our results suggest that the vulnerability finding-and-patching treadmill is not scalable. More effort should be invested in avoiding the introduction of vulnerabilities in the code in the first place, i.e. more 'security by design' efforts are needed (as already stated in the previous section and further analyzed in Section 3.7).

- In Chapter 4 we introduced a way to measure vulnerability lifetimes. This is the first measurement approach that allows us to estimate lifetimes automatically and accurately. Using this approach we performed a large-scale empirical study on characteristics of vulnerability lifetimes focusing on differences between projects and time-dependent effects. The research question we tried to answer in this chapter was: *How long do vulnerabilities live in the code?* We measured that vulnerabilities remain in the code for long period of time, varying significantly between projects (e.g. ~2 years for Chromium vs. ~7 years for OpenSSL). Vulnerability lifetimes are also increasing over time as the age of the codebase increases.

  We identified metrics that can provide indications regarding software security quality, most notably (a) the spread between the average lifetime of vulnerabilities and the age of the code when the vulnerabilities are discovered, (b) the rate of increase/decrease of this spread. We interpreted our observation that for Chromium this spread increases as an indication of maturity and increasing quality. We also observed that the introduction of fuzzing had no noticeable impact on the lifetimes of memory vulnerabilities for the Linux kernel. More efforts are needed towards measuring the impact of automated approaches such as fuzzing on the security of codebases. Vulnerability lifetimes, as a metric along with our measurement method, can be used to explore such questions and possibly many others.

- In Chapter 5 we introduced the first methodology to create a large dataset of vulnerability reporters for FLOSS from publicly available sources. Via a large-scale process of scraping, combining and cleaning data from a variety

of sources we were able to collect information on the people or organizations credited with reporting vulnerabilities in popular FLOSS projects. With this dataset we performed an empirical study on characteristics of vulnerability reporters focusing on differences between projects, developments over time and motivations of reporters. The main research question we tried to answer in this chapter was: *How can we measure human effort in the vulnerability discovery process?* This question aims to capture aspects of vulnerability reporters, their characteristics (affiliations, duration of engagement, etc.) that can be used to assess the health of a project's vulnerability-finding ecosystem.

We found that the number of reporters, as well as the number of new reporters per year were closely correlated with the number of vulnerability discoveries. Although there is correlation between the number of reports per reporter (the lower the better) and the collective effort invested in the vulnerability discovery process, we should not rush into using this metric to compare software. As we showed, for some cases as the Linux kernel, reporters often are big companies and organizations that might contribute hundreds of individuals as well as heavy machinery in the discovery effort. A complete overview of the reporting ecosystem of projects, following the methodology presented in this chapter and taking into account the parameters of Table 5.7, could be used to qualitatively compare the health of different projects. In light of new bug bounty programs targeting FLOSS, further research on the motivations of reporters and their development over time would provide interesting insights.

6.2 FURTHER DISCUSSION

Is software improving? – Is vulnerability hunting socially beneficial? In Chapter 3 we observed that there is no observable decreasing trend in the vulnerability discovery rate during Debian stable releases, while in Chapter 4 we observed that vulnerability lifetimes are increasing. These observations point to practical non-depletion of vulnerabilities, i.e. vulnerabilities are seemingly infinite, and thus finding and fixing them may not reduce the available attack surface for adversaries. However, in Chapter 4 we also observed that for some projects there are signs of maturity in the sense that the lifetime of vulnerabilities grows slower that the increase of the average age of the codebase, meaning that vulnerabilities are getting rarer in some old parts of the code. In the same chapter we also observed that vulnerability lifetimes are distributed exponentially, albeit with high half-lives. As we explain in Section 4.8 this would translate in an expected exponential decrease in the vulnerability discovery rate affecting a stable release assuming that the vulnerability discovery rate for a project is constant. The non-decreasing trend observed in Debian stable releases can be explained by an increasing trend in the discovery rate overall (affecting the newest version). The cause of this increase seems to be an increase in the effort expended in the vulnerability finding (and patching) processes. In Chapter 5 we indeed observed that the number of reporters per year – which can act as a proxy for the number of people searching for vulnerabilities – is correlated with

the number of vulnerability reports. Overall, we can say that although vulnerabilities may not seem to be depleted at the moment, and therefore the social utility of vulnerability finding may still be contested (echoing the concerns of Rescorla [93]), the observation that vulnerability lifetimes seem to grow slower than the age of the code suggests that at least for some parts of the code vulnerabilities are getting more difficult to find and therefore finding and fixing them is useful. We therefore believe that we will observe more definitive signs of depletion in the near future.

Practical impact of the contributions Although our contributions are largely fundamental, improving our understanding of security and introducing new avenues to measure important aspects affecting software quality, they can also find immediate practical impact. For example, vulnerability lifetimes (introduced in Chapter 4) can be used in case studies to measure and compare the effectiveness of different code review approaches as well as the impact of automated tools. Furthermore, such measurements can be used to support decision-making regarding the duration of long term support, as discussed in Section 4.8. Our main claims regarding seeming non-depletion of vulnerabilities support the need for more investment in "security by design" solutions.

6.3 FUTURE WORK

The core contributions of this dissertation are methods to measure quantities that can help us assess the security of software. These methods are of general applicability, and therefore can be utilized to investigate additional research questions. Here we provide an overview of possible avenues for future work.

More detailed per project analysis. In all three empirical studies presented in this dissertation we focused on a selection of popular FLOSS projects with a significant number of reported security vulnerabilities. However, small projects with few reported vulnerabilities also contribute to the overall security landscape and since statistical measures are difficult to apply in such cases (due to the low number of reports), other approaches are necessary (e.g. based on code analysis).

Further studies on the impact of automated tools. In Section 4.7 we included a small-scale case study showing how vulnerability lifetimes can be used to better understand and quantify the impact of automated vulnerability discovery tools (e.g. fuzzing) on the overall security of codebases. Further studies on the matter using vulnerability lifetimes or other metrics could lead to very interesting results. It is still an open question whether we can achieve better security by getting better in finding vulnerabilities or whether more fundamental changes in the software development process (e.g. using memory-safe programming languages) are necessary. Our findings (esp. the "Tip of the iceberg" observation of Chapter 3) support the second claim.

Further steps towards quantifying vulnerability discovery effort. In Chapter 5 we introduced an approach to perform large-scale studies on vulnerability reporters that enabled us to draw some interesting conclusions. However, we are

still some way off from assessing the effort (in terms of time, talent, equipment, money) that led to the discovery of a vulnerability. One way to achieve progress in this front would be for vulnerability reporters to include additional information in their reports that would help us make an automatic assessment of the effort invested.

Further studies on bug bounty programs for FLOSS. In light of new programs recently launched targeting FLOSS (e.g. by Google[1]) that offer more competitive bug bounties (compared to proprietary software), it would be interesting to revisit the bounty-related questions discussed in Chapters 3 and 5 in the near future.

Formalization efforts. As we improve our understanding of the vulnerability finding process and identify indicators that help us assess the quality of security of projects, a more formal expression of the models that underlie the process would be highly beneficial and interesting avenue for future work on the fundamental aspects of the problem.

Improvements in vulnerability tracing. Our dataset, method and observations regarding the estimation of the introduction date of a vulnerability (Chapter 4) could be used to create better tools for vulnerability tracing (tracing the commits that introduced a vulnerability). Further work could include the use of machine learning algorithms and advanced code representations (e.g. Code Property Graphs [116]).

1 https://security.googleblog.com/2020/05/expanding-our-work-with-open-source.html

A

COMPLETE LIST OF OWN PUBLICATIONS

1. Alexopoulos, N. *On enhancing trust in cryptographic solutions: student research abstract* in *Proceedings of the Symposium on Applied Computing, SAC 2017, Marrakech, Morocco, April 3-7, 2017* (eds Seffah, A., Penzenstadler, B., Alves, C. & Peng, X.) (ACM, 2017), 1848–1849.

2. Alexopoulos, N., Brack, M., Wagner, J. P., Grube, T. & Mühlhäuser, M. *How long do vulnerabilities live in the code? A large-scale empirical measurement study on FOSS vulnerability lifetimes* in *31st USENIX Security Symposium, USENIX Security 2022 (to appear)* (USENIX Association, 2022).

3. Alexopoulos, N., Daubert, J., Mühlhäuser, M. & Habib, S. M. *Beyond the Hype: On Using Blockchains in Trust Management for Authentication* in *2017 IEEE Trustcom/BigDataSE/ICESS, Sydney, Australia, August 1-4, 2017* (IEEE Computer Society, 2017), 546–553.

4. Alexopoulos, N., Egert, R., Grube, T. & Mühlhäuser, M. *Poster: Towards Automated Quantitative Analysis and Forecasting of Vulnerability Discoveries in Debian GNU/Linux* in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019* (eds Cavallaro, L., Kinder, J., Wang, X. & Katz, J.) (ACM, 2019), 2677–2679.

5. Alexopoulos, N., Habib, S. M. & Mühlhäuser, M. *Towards Secure Distributed Trust Management on a Global Scale: An analytical approach for applying Distributed Ledgers for authorization in the IoT* in *Proceedings of the 2018 Workshop on IoT Security and Privacy, IoT S&P@SIGCOMM 2018, Budapest, Hungary, August 20, 2018* (ACM, 2018), 49–54.

6. Alexopoulos, N., Habib, S. M., Schulz, S. & Mühlhäuser, M. The Tip of the Iceberg: On the Merits of Finding Security Bugs. *ACM Trans. Priv. Secur.* **24** (2020).

7. Alexopoulos, N., Kiayias, A., Talviste, R. & Zacharias, T. *MCMix: Anonymous Messaging via Secure Multiparty Computation* in *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017* (eds Kirda, E. & Ristenpart, T.) (USENIX Association, 2017), 1217–1234.

8. Alexopoulos, N., Meneely, A., Arnouts, D. & Mühlhäuser, M. *Who are Vulnerability Reporters?: A Large-scale Empirical Study on FLOSS* in *ESEM '21: ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, Bari, Italy, October 11-15, 2021* (eds Lanubile, F., Kalinowski, M. & Baldassarre, M. T.) (ACM, 2021), 25:1–25:12.

9. Alexopoulos, N., Vasilomanolakis, E., Ivánkó, N. R. & Mühlhäuser, M. *Towards Blockchain-Based Collaborative Intrusion Detection Systems* in *Critical Information Infrastructures Security - 12th International Conference, CRITIS 2017, Lucca, Italy, October 8-13, 2017, Revised Selected Papers* (eds D'Agostino, G. & Scala, A.) **10707** (Springer, 2017), 107–118.

10. Alexopoulos, N., Vasilomanolakis, E., Roux, S. L., Rowe, S. & Mühlhäuser, M. *TRIDEnT: towards a decentralized threat indicator marketplace* in *SAC '20: The 35th ACM/SIGAPP Symposium on Applied Computing, online event, [Brno, Czech Republic], March 30 - April 3, 2020* (eds Hung, C., Cerný, T., Shin, D. & Bechini, A.) (ACM, 2020), 332–341.

11. Böck, L., Alexopoulos, N., Saracoglu, E., Mühlhäuser, M. & Vasilomanolakis, E. *Assessing the Threat of Blockchain-based Botnets* in *2019 APWG Symposium on Electronic Crime Research, eCrime 2019, Pittsburgh, PA, USA, November 13-15, 2019* (IEEE, 2019), 1–11.

12. Habib, S. M., Alexopoulos, N., Islam, M. M., Heider, J., Marsh, S. & Mühlhäuser, M. *Trust4App: Automating Trustworthiness Assessment of Mobile Applications* in *17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications / 12th IEEE International Conference On Big Data Science And Engineering, TrustCom/BigDataSE 2018, New York, NY, USA, August 1-3, 2018* (IEEE, 2018), 124–135.

Appendices

B

APPENDIX OF CHAPTER 3

B.1 ADDITIONAL FIGURES

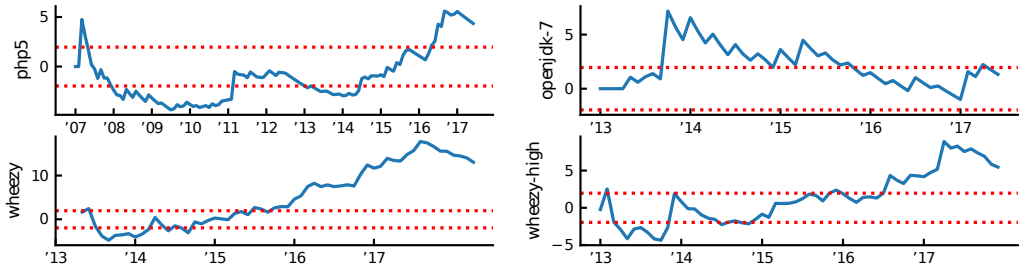


Figure B.1: Laplace trend tests with 95% significance thresholds (dashed lines).

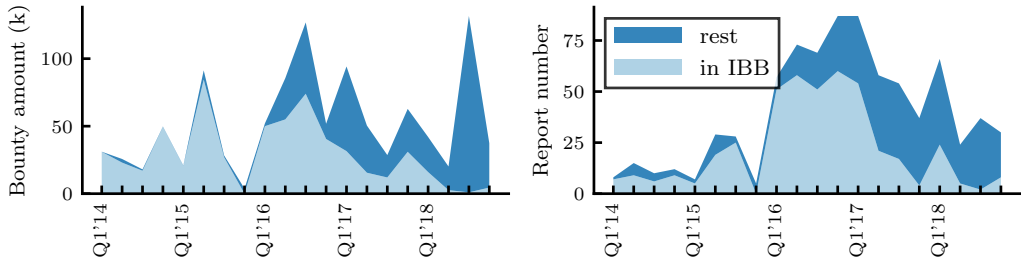


Figure B.2: Bounty amounts in thousands of USD (left) and number of reports (right) of IBB reporters (at least 1 IBB report at some point in time) comparing reports in the IBB program against reports for other programs in HackerOne over time.

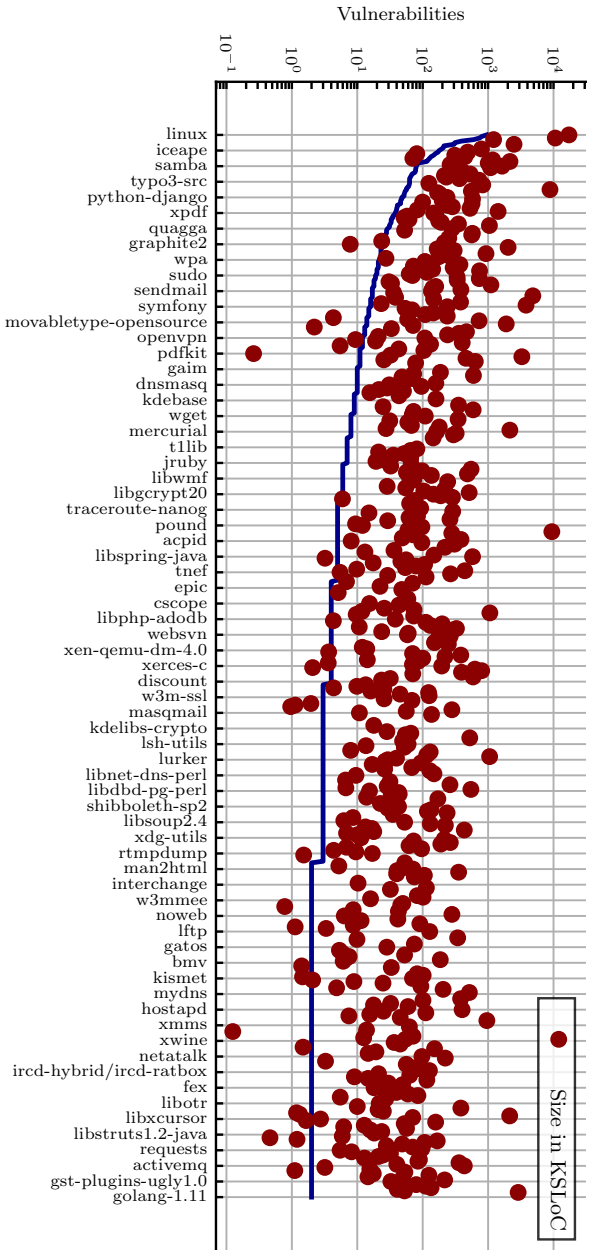


Figure B.3: The distribution of vulnerabilities in the Debian ecosystem (years 2001-2017), along with the size of the corresponding packages. The scale of axis x is logarithmic. All packages are taken into account. Every tenth package name appears on the y axis for space reasons.

B.2 STATISTICAL TEST RESULTS

Detailed statistical test results referring to the plots of the chapter's main body are included in this section.

Table B.1: OLS for reliability trends

Trend of the total number of vulnerabilities in Debian (Fig. 3.5a).

Dep. Variable:	total	**R-squared:**	0.796
Model:	OLS	**Adj. R-squared:**	0.783
Method:	Least Squares	**F-statistic:**	62.36
No. Observations:	18	**AIC:**	234.7
Df Residuals:	16	**BIC:**	236.5
Df Model:	1		

	coef	std err	t	P>|t|	[0.025	0.975]
const	120.0175	70.519	1.702	0.108	-29.476	269.511
x1	55.9195	7.081	7.897	0.000	40.907	70.932

Omnibus:	6.195	**Durbin-Watson:**	1.284
Prob(Omnibus):	0.045	**Jarque-Bera (JB):**	3.587
Skew:	0.994	**Prob(JB):**	0.166
Kurtosis:	3.910	**Cond. No.**	19.3

Trend of the average number of vulnerabilities in Debian (Fig. 3.5b).

Dep. Variable:	av. per package	**R-squared:**	0.919
Model:	OLS	**Adj. R-squared:**	0.914
Method:	Least Squares	**F-statistic:**	180.7
No. Observations:	18	**AIC:**	28.43
Df Residuals:	16	**BIC:**	30.21
Df Model:	1		

	coef	std err	t	P>|t|	[0.025	0.975]
const	1.1114	0.229	4.856	0.000	0.626	1.597
x1	0.3089	0.023	13.442	0.000	0.260	0.358

Omnibus:	3.018	**Durbin-Watson:**	1.680
Prob(Omnibus):	0.221	**Jarque-Bera (JB):**	1.300
Skew:	0.594	**Prob(JB):**	0.522
Kurtosis:	3.566	**Cond. No.**	19.3

Trend of the number of vulnerabilities in Debian Wheezy (Fig. 3.8).

Dep. Variable:	Wheezy total	**R-squared:**	0.564
Model:	OLS	**Adj. R-squared:**	0.540
Method:	Least Squares	**F-statistic:**	23.29
No. Observations:	20	**AIC:**	223.7
Df Residuals:	18	**BIC:**	225.7
Df Model:	1		

	coef	std err	t	P>|t|	[0.025	0.975]
const	153.1143	26.723	5.730	0.000	96.972	209.256
x1	11.6038	2.405	4.826	0.000	6.552	16.656

Omnibus:	1.893	**Durbin-Watson:**	1.736
Prob(Omnibus):	0.388	**Jarque-Bera (JB):**	0.748
Skew:	0.445	**Prob(JB):**	0.688
Kurtosis:	3.324	**Cond. No.**	21.5

Trend of the number of high-severity vulnerabilities in Debian Wheezy (Fig. 3.12).

Dep. Variable:	Wheezy high	**R-squared:**	0.318
Model:	OLS	**Adj. R-squared:**	0.280
Method:	Least Squares	**F-statistic:**	8.402
No. Observations:	20	**AIC:**	192.1
Df Residuals:	18	**BIC:**	194.1
Df Model:	1		

	coef	std err	t	P>|t|	[0.025	0.975]
const	46.8571	12.104	3.871	0.001	21.427	72.287
x1	3.1571	1.089	2.899	0.010	0.869	5.445

Omnibus:	20.152	**Durbin-Watson:**	1.990
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	26.851
Skew:	1.753	**Prob(JB):**	1.48e-06
Kurtosis:	7.464	**Cond. No.**	21.5

Table B.2: OLS for bug bounty trends (Fig. 3.17)

Trend of average price in the IBB program.

Dep. Variable:	IBB-all-av	**R-squared:**	0.245
Model:	OLS	**Adj. R-squared:**	0.200
Method:	Least Squares	**F-statistic:**	5.513
No. Observations:	19	**AIC:**	343.1
Df Residuals:	17	**BIC:**	345.0
Df Model:	1		

	coef	std err	t	P>|t|	[0.025	0.975]
const	3918.9048	845.666	4.634	0.000	2134.706	5703.104
x1	-175.8950	74.917	-2.348	**0.031**	-333.955	-17.835

Omnibus:	26.771	**Durbin-Watson:**	1.939
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	42.644
Skew:	2.326	**Prob(JB):**	5.50e-10
Kurtosis:	8.677	**Cond. No.**	21.8

Trend of median price in the IBB program.

Dep. Variable:	IBB-all-med	**R-squared:**	0.426
Model:	OLS	**Adj. R-squared:**	0.392
Method:	Least Squares	**F-statistic:**	12.60
No. Observations:	19	**AIC:**	309.7
Df Residuals:	17	**BIC:**	311.6
Df Model:	1		

	coef	**std err**	**t**	**P>|t|**	**[0.025**	**0.975]**
const	2365.1079	350.802	6.742	0.000	1624.980	3105.236
x1	-110.3118	31.077	-3.550	**0.002**	-175.879	-44.745

Omnibus:	6.741	**Durbin-Watson:**	1.636
Prob(Omnibus):	0.034	**Jarque-Bera (JB):**	4.340
Skew:	1.116	**Prob(JB):**	0.114
Kurtosis:	3.705	**Cond. No.**	21.8

Trend of average price in the IBB program - only high and critical severity bugs.

Dep. Variable:	IBB-high-av	**R-squared:**	0.023
Model:	OLS	**Adj. R-squared:**	-0.303
Method:	Least Squares	**F-statistic:**	0.06945
No. Observations:	5	**AIC:**	86.40
Df Residuals:	3	**BIC:**	85.62
Df Model:	1		

	coef	**std err**	**t**	**P>|t|**	**[0.025**	**0.975]**
const	2851.3514	4400.740	0.648	0.563	-1.12e+04	1.69e+04
x1	-81.0811	307.661	-0.264	0.809	-1060.197	898.034

Omnibus:	nan	**Durbin-Watson:**	3.543
Prob(Omnibus):	nan	**Jarque-Bera (JB):**	0.406
Skew:	0.242	**Prob(JB):**	0.816
Kurtosis:	1.691	**Cond. No.**	119.

Trend of median price in the IBB program - only high and critical severity bugs.

Dep. Variable:	IBB-high-med	**R-squared:**	0.023
Model:	OLS	**Adj. R-squared:**	-0.303
Method:	Least Squares	**F-statistic:**	0.06945
No. Observations:	5	**AIC:**	86.40
Df Residuals:	3	**BIC:**	85.62
Df Model:	1		

	coef	std err	t	P>|t|	[0.025	0.975]
const	2851.3514	4400.740	0.648	0.563	-1.12e+04	1.69e+04
x1	-81.0811	307.661	-0.264	0.809	-1060.197	898.034

Omnibus:	nan	**Durbin-Watson:**	3.543
Prob(Omnibus):	nan	**Jarque-Bera (JB):**	0.406
Skew:	0.242	**Prob(JB):**	0.816
Kurtosis:	1.691	**Cond. No.**	119.

Trend of average price in HackerOne.

Dep. Variable:	all-all-av	**R-squared:**	0.185
Model:	OLS	**Adj. R-squared:**	0.140
Method:	Least Squares	**F-statistic:**	4.094
No. Observations:	20	**AIC:**	278.6
Df Residuals:	18	**BIC:**	280.6
Df Model:	1		

	coef	std err	t	P>|t|	[0.025	0.975]
const	614.9606	105.436	5.833	0.000	393.448	836.473
x1	19.1973	9.488	2.023	0.058	-0.735	39.130

Omnibus:	15.357	**Durbin-Watson:**	1.291
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	14.439
Skew:	1.621	**Prob(JB):**	0.000732
Kurtosis:	5.611	**Cond. No.**	21.5

Trend of median price in HackerOne.

Dep. Variable:	all-all-med	**R-squared:**	0.222
Model:	OLS	**Adj. R-squared:**	0.179
Method:	Least Squares	**F-statistic:**	5.141
No. Observations:	20	**AIC:**	243.8
Df Residuals:	18	**BIC:**	245.8
Df Model:	1		

	coef	std err	t	P>|t|	[0.025	0.975]
const	241.2714	44.100	5.471	0.000	148.622	333.921
x1	8.9977	3.968	2.267	**0.036**	0.661	17.335

Omnibus:	7.109	**Durbin-Watson:**	1.723
Prob(Omnibus):	0.029	**Jarque-Bera (JB):**	4.967
Skew:	1.189	**Prob(JB):**	0.0834
Kurtosis:	3.550	**Cond. No.**	21.5

Trend of average price in HackerOne – only high and critical severity bugs.

Dep. Variable:	all-high-av	**R-squared:**	0.213
Model:	OLS	**Adj. R-squared:**	0.134
Method:	Least Squares	**F-statistic:**	2.707
No. Observations:	12	**AIC:**	217.0
Df Residuals:	10	**BIC:**	218.0
Df Model:	1		

	coef	**std err**	**t**	**P>|t|**	**[0.025**	**0.975]**
const	5446.5080	1574.979	3.458	0.006	1937.236	8955.780
x1	-191.7995	116.585	-1.645	0.131	-451.567	67.969

Omnibus:	1.989	**Durbin-Watson:**	2.116
Prob(Omnibus):	0.370	**Jarque-Bera (JB):**	0.237
Skew:	-0.093	**Prob(JB):**	0.888
Kurtosis:	3.663	**Cond. No.**	39.0

Trend of median price in HackerOne – only high and critical severity bugs.

Dep. Variable:	all-high-med	**R-squared:**	0.282
Model:	OLS	**Adj. R-squared:**	0.210
Method:	Least Squares	**F-statistic:**	3.922
No. Observations:	12	**AIC:**	222.2
Df Residuals:	10	**BIC:**	223.2
Df Model:	1		

	coef	**std err**	**t**	**P>|t|**	**[0.025**	**0.975]**
const	6053.7931	1957.525	3.093	0.011	1692.157	1.04e+04
x1	-286.9574	144.902	-1.980	0.076	-609.820	35.905

Omnibus:	6.127	**Durbin-Watson:**	2.031
Prob(Omnibus):	0.047	**Jarque-Bera (JB):**	2.495
Skew:	0.937	**Prob(JB):**	0.287
Kurtosis:	4.216	**Cond. No.**	39.0

C

APPENDIX OF CHAPTER 4

C.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

This is by no means a complete mapping, but covers all CWEs represented in our dataset. Please note that CWE-NVD-noinfo is not assigned a category.

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: np.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}

C.2 TRENDS

Statistical summary for vulnerability lifetime trends for all projects with more than two datapoints are listed in tables C.1 - C.7.

	coef	std err	t	P> |t|	[0.025	0.975]
const	923.75	90.62	10.19	0.062	-227.74	2075.24
x1	45.05	20.61	2.18	0.27	-216.83	306.93

Table C.1: OLS Regression Results: FFmpeg

	coef	std err	t	P> |t|	[0.025	0.975]
const	-3e+05	5e+04	-5.62	0.005	-4.64e+05	-1.57e+05
x1	155.61	27.51	5.65	0.005	79.22	232.00

Table C.2: OLS Regression Results: Httpd

	coef	std err	t	P> |t|	[0.025	0.975]
const	-8e+04	2.5e+04	-3.28	0.01	-1.4e+05	-2.6e+04
x1	42.23	12.77	3.30	0.009	13.34	71.13

Table C.3: OLS Regression Results: Chromium

	coef	std err	t	P> |t|	[0.025	0.975]
const	2.2e+04	2.6e+04	0.82	0.43	-3.7e+04	8.1e+04
x1	-10.27	13.35	-0.76	0.45	-39.66	19.11

Table C.4: OLS Regression Results: Firefox

	coef	std err	t	P> |t|	[0.025	0.975]
const	-3.0e+05	3.4e+04	-8.93	0.00	-3e+05	-2e+05
x1	153.20	16.94	9.04	0.00	114.12	192.27

Table C.5: OLS Regression Results: Linux

	coef	std err	t	P> |t|	[0.025	0.975]
const	-5.4e+05	2.4e+05	-2.23	0.15	-1.6e+06	5.e+05
x1	273.41	121.91	2.24	0.15	-251.15	797.98

Table C.6: OLS Regression Results: Wireshark

	coef	std err	t	P> |t|	[0.025	0.975]
const	-2.2e+05	1.4e+05	-1.62	0.20	-6.7e+05	2.1e+05
x1	114.08	69.54	1.64	0.19	-107.24	335.42

Table C.7: OLS Regression Results: Qemu

C.3 LIFETIME DISTRIBUTION

Evaluation of exponential fit. Prior seminal work on fitting theoretical distributions to noisy empirical data in computer science (e.g. Stutzbach and Re-

jaie [105]) used suitable plots (CCDF in logarithmic and linear axes) to assess the goodness of fit of a given type of distribution. They only used statistical goodness of fit tests, like the Anderson-Darling statistic, in cases where only a few hundred data points were available.

As Johnson and Wichern state in their seminal statistical analysis textbook [47]: "all measures of goodness of fit suffer the same serious drawback[.] When the sample size is small, only the most aberrant behavior will be identified as lack of fit. On the other hand, very large samples invariably produce statistically significant lack of fit. Yet the departure from the specified distribution may be very small and technically unimportant to the inferential conclusions". For example, Wicklin showed[1] that even rounding to the nearest 0.1 unit can cause a normality test to fail with 5 000 data points.

For our case, expecting the empirical data of more than 5 000 vulnerability lifetimes to plausibly be produced (as in statistical goodness-of-fit tests) by random sampling of a simple theoretical distribution (such as an exponential) would be unreasonable. Indeed, all such tests showed that the probability that the empirical data come from an exponential distribution is negligible. We attribute this to the large number of data points, inherent noise in the data, and the differing behavior at the tail (as discussed below).

This does not mean that the exponential distribution cannot be an excellent fit to the data for all purposes of reasoning about or making calculations on vulnerability lifetimes. We continue to provide evidence of a good fit by (a) a Q-Q plot, (b) Kolmogorov-Smirnov statistical tests comparing the fit to other candidate distributions, (c) example numerical calculations showcasing the utility of the fitted distribution.

Kolmogorov-Smirnov tests. We list the results of the comparison using the seminal methodology of Clauset et al. [24] in Table C.8. 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 C.8: 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% ($p \leqslant 0.01$).

Comparative probability table Table C.9 is a comparative probability table to numerically convey the goodness-of-fit of the exponential distribution and its usefulness in calculations.

1 https://blogs.sas.com/content/iml/2016/11/28/goodness-of-fit-large-small-samples.html

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 C.9: 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.

C.4 PROJECT-SPECIFIC DETAILS ON MAPPING CVES TO THEIR FIXING COMMITS

Here we document the resulting number of obtained mappings between CVEs and fixing commits per matching approach. Please note that each mapping is only counted once and numbers correspond to the mapped fixing commits (right column of Table 4.2). Mappings appearing in multiple of the sources are counted towards the first one listed. For more details on the specific regular expressions for each project please refer to the source code[2].

- **Chromium**:

Common Bug ID	2506
Vulnerability History Project	331

- **FFmpeg**:

Commit Sha in CVE	208
Debian Security Tracker	98
CVE in Fixing commmit	68
Manual inspection	1

- **Firefox**:

Common Bug ID	3751

- **Httpd**:

Piantadosi Apache	378
Vulnerability History Project	98

2 https://github.com/nikalexo/VulnerabilityLifetimes

- **kernel**:

Linux Kernel CVES	1341
Ubuntu CVE Tracker	190

- **OpenSSL**:

CVE in Fixing commmit	145
Commit Sha in CVE	80
Debian Security Tracker	35
Manual inspection	1

- **php**:

Common Bug ID	901
Commit Sha in CVE	31

- **postgres**:

CVE in Fixing commmit	133
Debian Security Tracker	9
Manual inspection	1

- **qemu**:

Debian Security Tracker	161
Commit Sha in CVE	131

- **tcpdump**:

Commit Sha in CVE	123
Debian Security Tracker	5

- **wireshark**:

Commit Sha in CVE	113
Common Bug ID	1

C.5 ADDITIONAL FIGURES

Figures C.1 - C.6 depict the vulnerability lifetime and regular code age for the projects not presented in the main body of the document. We did not have enough data points to conduct this analysis for TCPDump, which is therefore omitted.

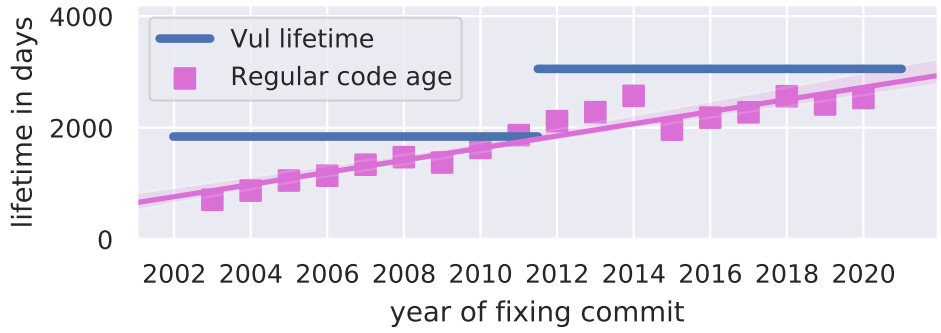


Figure C.1: Regular code and vulnerability age for PHP. Vulnerable code seems to be older than regular code for this project.

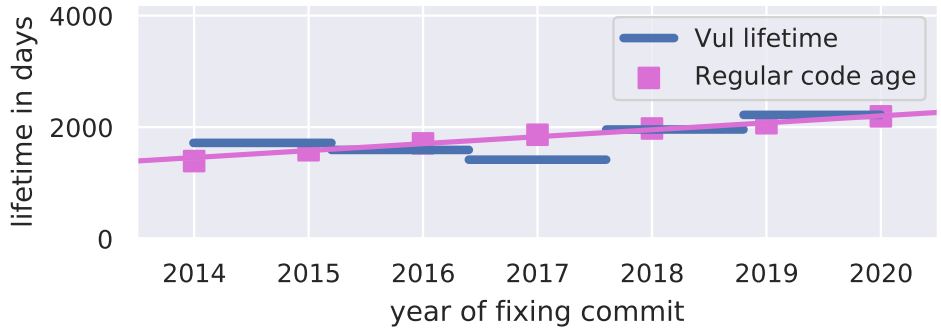


Figure C.2: Regular code and vulnerability age for qemu

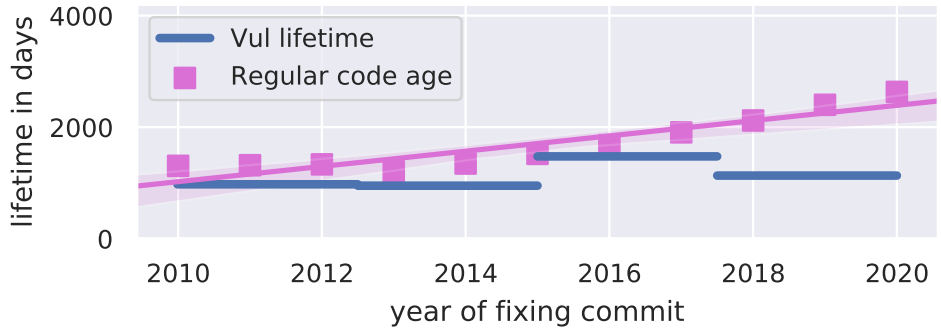


Figure C.3: Regular code and vulnerability age for ffmpeg

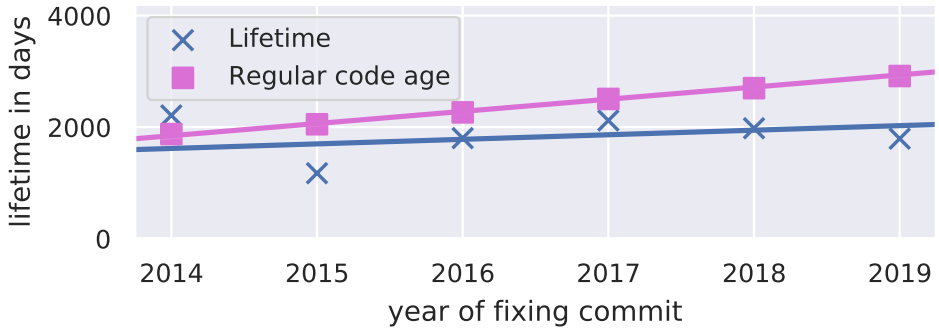


Figure C.4: Regular code and vulnerability age for wireshark

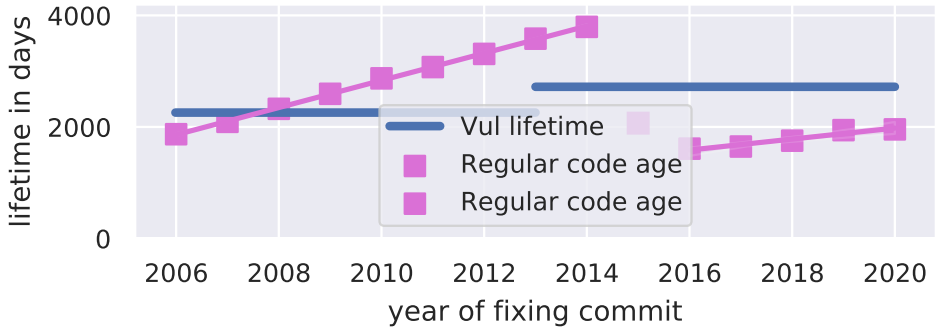


Figure C.5: Regular code and vulnerability age for openssl. The refactoring process that took place after the Heartbleed bug is evident. A shortage of data points does not allow us to investigate its effect in detail.

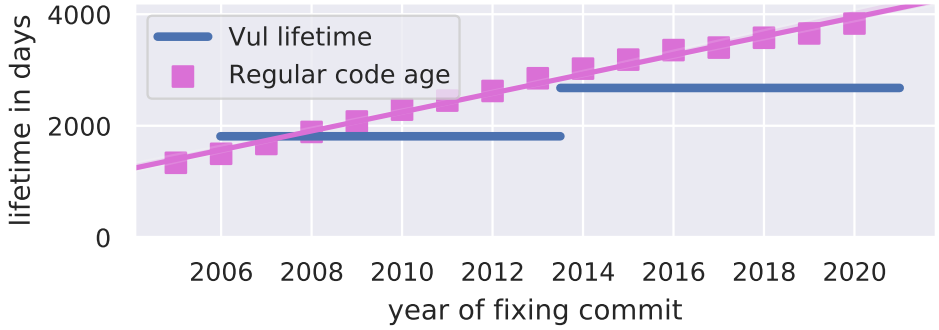


Figure C.6: Regular code and vulnerability age for postgres

C.6 MANUAL ANALYSIS OF VULNERABILITY-CONTRIBUTING COMMITS

C.6.1 *Introduction*

During our research with Jan Wagner and Manuel Brack, we noticed some discrepancies between Vulnerability-Contributing-Commits (VCCs) as reported by the VCCFinder heuristic and the Ubuntu Kernel Security Team[3]. Ten of those disputed CVEs were randomly picked for further manual analysis. The picked CVEs, along with their fixing and reported VCCs, are listed in the following table.

CVE	Fix Ubuntu Security Team VCCFinder
CVE-2014-5206	a6138db815df5ee542d848318e5dae681590fccd 0c55cfc4166d9a0f38de779bd4d75a90afbe7734 495d6c9c6595ec7b37910dfd42634839431d21fd
CVE-2013-4127	dd7633ecd553a5e304d349aa6f8eb8a0417098c5 1280c27f8e29acf4af2da914e80ec27c3dbd5c01 283940of8fe28ce216eeeba3fb97bdf90977f7ad
CVE-2013-1819	eb178619f930fa2ba2348de332a1ff1c66a31424 74f75a0cb7033918eb0fa4a50df25091ac75c16e 3e85c868a6978o5a3d4c7800a6bacdfc81d15cdf
CVE-2013-7348	d558023207e008a4476a3b7bb8706b2a2bf5d84f e34ecee2ae791df674dfb466ce40692ca6218e43 e23754f880f10124f0a2848f9d17e361a295378e
CVE-2014-5045	295dc39d941dc2ae53d5c170365af4c9d5c16212 8033426e6bdb2690d302872ac1e1fadaec1a5581 35759521eedf60ce7d3127c5d33953cd2d1bd35f
CVE-2012-1097	c8e252586f8d5de906385d8cf6385fee289a825e bdf88217b70dbb18c4ee27a6c497286e040a6705 5bde4d181793be84351bc21c256d8c71cfcd313a
CVE-2014-0196	4291086b1f081b869c6d79e5b7441633dc3ace00 d945cb9cce20ac7143c2de8d88b187f62db99bdc d6afe27bfff30fbec2cca6ad5626c22f4094d770
CVE-2013-1979	83f1b4ba917db5dc5a061a44b3403ddb6e783494 257b5358b32f17e0603b6ff57b13610b0e02348f dbe9a4173ea53b72b2c35d19f676a85b69f1c9fe
CVE-2011-1479	dode4dc584ec6aa3b26fffea320a8457827768fc a2ae4cc9a16e211c8a128ba10d22a85431f093ab 2d9048e201bfb67ba21f05e647b1286b8a4a5667
CVE-2013-4470	c547dbf55d5f8cf615ccc0e7265e98db27d3fb8b e89e9cf539a28df7d0eb1d0a545368e9920b34ac c31d5326902cebffcd83b1aede67a0e0ac923090

3 The content of this section was first published in the Appendix of Jan Wagner's Bachelor's thesis (https://fileserver.tk.informatik.tu-darmstadt.de/NA/Thesis_JW.pdf).

C.6.2 *Analysis*

Results of the manual analysis follow.

C.6.2.1 *CVE-2014-5206*

Date: 08/18/2014

Type: CWE-264 – Permissions, Privileges, and Access Controls
Description: The do_remount function in fs/namespace.c in the Linux kernel through 3.16.1 does not maintain the MNT_LOCK_READONLY bit across a remount of a bind mount, which allows local users to bypass an intended read-only restriction and defeat certain sandbox protection mechanisms via a "mount -o remount" command within a user namespace.
Notes: VCCFinder blames a commit that cleans up code. It changes the expression for the mount flag from a literal constant to a bitwise operation on constants defined in the header file. No functionality difference should exist.

Reason for discrepancy: This is an interesting one. It seems that the vulnerability is introduced in the commit blamed by the Ubuntu security team that allows non-root users to perform operations on namespaces. Before then, there was no reason to protect the flags. Specifically in the SYSCALL definition of umount.

Verdict: The commit blamed by the Ubuntu teams seems to be the VCC. It seems very difficult for a blame-based heuristic to pinpoint this commit. The VCC pinpointed by VCCFinder is definitely wrong though, as it is just refactoring. Namespaces are a very high-risk location for those kind of vulnerabilities. This VCC is also responsible for the related CVE-2014-5207. Interestingly, these vulnerabilities did not affect the Debian stable distribution, where by default user namespaces are disabled. Commits that change a lot of if statements that affect system calls should be considered high-risk, especially when they advertise introducing intrinsically high-risk functionalities. Maybe a good recommendation would be, when dealing with access control bugs to look more into changes to if statements. Another recommendation would be to cosider such files (i.e. files that regulate access) as high-risk. Another observation is that such kinds of semantic flaws can only be automatically found when metadata, such as the commit messages are available.

C.6.2.2 *CVE-2013-4127*

Date: 07/29/2013

Type: CWE-399 – Resource Management Errors

Description: Use-after-free vulnerability in the vhost_net_set_backend function in drivers/vhost/net.c in the Linux kernel through 3.10.3 allows local users to cause a denial of service (OOPS and system crash) via vectors involving powering on a virtual machine.

Notes: This is a use-after-free vulnerability and thus should be able to be detected automatically pretty easily. The VCC proposed by VCCFinder is not correct. It seems to only be copying code from one file to another. The commit

proposed by the Ubuntu security team seems to be the correct VCCs. This is also noted in the commit message of the fix.

Reason for discrepancy: It is impossible to trace code back to its original file after it has been copied to another file. That being said, in this specific case, the vulnerability was introduced in the file that the code was copied to.

Verdict: The VCC of the Ubuntu Security Team is correct. If a heuristic finds code copying, then maybe it would make sense to look into previous commits of the source file. Naturally, for a use-after-free vulnerability, we can expect the VCC to free memory.

C.6.2.3 *CVE-2013-1819*

Date: 03/06/2013

Type:CWE-20 – Improper Input Validation

Description: The _xfs_buf_find function in fs/xfs/xfs_buf.c in the Linux kernel before 3.7.6 does not validate block numbers, which allows local users to cause a denial of service (NULL pointer dereference and system crash) or possibly have unspecified other impact by leveraging the ability to mount an XFS filesystem containing a metadata inode with an invalid extent map.

Notes: The correct commit is also blamed by the VCCFinder heuristic but is not the most dominant.

Reason for discrepancy: Empty line contributes to the mistake. Also, blaming where the declaration of variables takes place could generally be bad practice.

Verdict: The Ubuntu Security Team seems to have the correct commit. VCCFinder produces a wrong commit, however we can learn from this mistake.

C.6.2.4 *CVE-2013-7348*

Date: 04/01/2014

Type: CWE-399 – Resource Management Errors

Description: Double free vulnerability in the ioctx_alloc function in fs/aio.c in the Linux kernel before 3.12.4 allows local users to cause a denial of service (system crash) or possibly have unspecified other impact via vectors involving an error condition in the aio_setup_ring function.

Notes: A double-free vulnerability can be easily detected by automated tools. We expect to blame the commit that introduces the 2nd free. In this case, it is more complicated than that. The vulnerability is introduced by the Ubuntu commit. This commit does not add a new free command. It changes the point where the error handling code "goes-to".

Reason for discrepancy: The fix was to remove one of the free instructions. The more general free instruction was removed, however the vulnerability was introduced in the more specific function. The phrase in the commit message: "clean up ioctx_alloc()'s error path" is what points to the problem.

Verdict: The Ubuntu commit is correct, however it was not trivial at all to see. Go-to in error handling seems to be dangerous. It is difficult to see how a blame-based heuristic would find the right commit.

C.6.2.5 *CVE-2014-5045*

Date: 08/01/2014

Type: CWE-59 – Improper Link Resolution Before File Access ('Link Following')

Description: The mountpoint_last function in fs/namei.c in the Linux kernel before 3.15.8 does not properly maintain a certain reference count during attempts to use the umount system call in conjunction with a symlink, which allows local users to cause a denial of service (memory consumption or use-after-free) or possibly have unspecified other impact via the umount program.

Notes: This is simple. The Ubuntu-blamed commit created the file and introduced the vulnerability.

Reason for discrepancy: The VCCFinder commit, as its message says, is massaging the code.

Verdict: The Ubuntu commit is the correct one. The VCCFinder commit just refactored the code in the area. Maybe going 2 steps back in time would have been beneficial. It does not seem very difficult to find this VCC automatically. The patterns existed before.

C.6.2.6 *CVE-2012-1097*

Date: 05/17/2012

Type: NVD-CWE-Other

Description: The regset (aka register set) feature in the Linux kernel before 3.2.10 does not properly handle the absence of .get and .set methods, which allows local users to cause a denial of service (NULL pointer dereference) or possibly have unspecified other impact via a (1) PTRACE_GETREGSET or (2) PTRACE_SETREGSET ptrace call.

Notes: This is a strange one. The two commits blamed by the Ubuntu team and VCCFinder were made the same day by the same person. The VCCFinder commit just adds some code to the first commit. This vulnerability in general has to do with error-checking. The reason behind it is that some developers did not follow what the expected behaviour of the functions were based on their header-file definitions. It is tough to blame a specific commit for such a vulnerability.

Reason for discrepancy: The VCCFinder commit adds some code to the earlier Ubuntu commit. Another possible commit to blame is:
4206d3aa1978e44f58bfa4e1c9d8d35cbf19c187.

Verdict: This vulnerability appeared when function implementations started diverging from the definitions in the header files. All 3 possible blame-able commits could be valid.

C.6.2.7 *CVE-2014-0196*

Date: 05/07/2014

Type: CWE-362 – Concurrent Execution using Shared Resource with Improper

Synchronization ('Race Condition')

Description: The n_tty_write function in drivers/tty/n_tty.c in the Linux kernel through 3.14.3 does not properly manage tty driver access in the "LECHO & !OPOST" case, which allows local users to cause a denial of service (memory corruption and system crash) or gain privileges by triggering a race condition involving read and write operations with long strings.

Notes: This is an interesting case. There was a lot of discussion in the Red Hat issue tracking system about this vulnerability[4]. Such kinds of concurrency bugs can be difficult to analyze. This bug seems to be caused by an update to the pty driver file, while the fix was introduced in the tty base driver file.

Reason for discrepancy: Concurrency bugs are sometimes difficult to handle. In this case it would have been better to blame the line in the middle of the added locks.

Verdict: The Ubuntu commit seems to be the correct one. There has been a lot of discussion about this bug.

C.6.2.8 *CVE-2013-1979*

Date: 05/03/2013

Type: CWE-264 – Permissions, Privileges, and Access Controls

Description: The scm_set_cred function in include/net/scm.h in the Linux kernel before 3.8.11 uses incorrect uid and gid values during credentials passing, which allows local users to gain privileges via a crafted application.

Notes: This is again a semantic bug. There is a clear mention in the fixing commit that the introducing commit is the Ubuntu one.

Reason for discrepancy: VCCFinder blames a commit that changed the code but not its functionality. The vulnerable code was in the called function body. The vulnerability was introduced 2 years before.

Verdict: The Ubuntu commit is the correct one. It seems very difficult to spot this VCC with blame heuristics. VCCFinder does a good job of blaming the specific lines, however the funtionality of the blamed lines existed in a function call from earlier. It remains to be seen if it is worth it to look for such behavior (i.e. look if one of the blamed lines is a function call and then look if the blamed code was part of the function body...). In general, the approach where after pinpointing a VCC some checks are carried out to see if we should go even further back in time (maybe with git dissect?) could be beneficial. This is a very difficult case for blame-based VCCs.

C.6.2.9 *CVE-2011-1479*

Date: 06/21/2012

Type: CWE-399 – Resource Management Errors

Description: Double free vulnerability in the inotify subsystem in the Linux kernel before 2.6.39 allows local users to cause a denial of service (system crash)

4 https://bugzilla.redhat.com/show_bug.cgi?id=1094232

via vectors involving failed attempts to create files. NOTE: this vulnerability exists because of an incorrect fix for CVE-2010-4250.

Notes: As noted in the description, this is a regression bug. VCCFinder blames a lot of older commits, since the changes in the fix affect a lot of lines. The most blamed commit is much older (4 years older than the real VCC).

Reason for discrepancy: The double-free is not direct. It happens via a function.

Verdict: The Ubuntu commit is the correct one. It seems very difficult for heuristics to find this regression bug, since the fix changes a lot of things.

C.6.2.10 *CVE-2013-4470*

Date: 11/04/2013

Type: CWE-264 – Permissions, Privileges, and Access Controls

Description: The Linux kernel before 3.12, when UDP Fragmentation Offload (UFO) is enabled, does not properly initialize certain data structures, which allows local users to cause a denial of service (memory corruption and system crash) or possibly gain privileges via a crafted application that uses the UDP_CORK option in a setsockopt system call and sends both short and long packets, related to the ip_ufo_append_data function in net/ipv4/ip_output.c and the ip6_ufo_append_data function in net/ipv6/ip6_output.c.

Notes: Another semantic bug (access control). There are 2 very similar fixing commits, one each for IPv4 and IPv6.

Reason for discrepancy: There is no discrepancy according to the latest results. Maybe some change to the heuristic improved performance.

Verdict: No discrepancy. The improvement seems to come from not taking into account the comments.

C.6.3 *Discussion*

- The Ubuntu commits are as close to a ground truth dataset as we can get
- Manual analysis is difficult and it seems previous research fell in the trap of wanting to confirm what the heuristic produced.
- Variable declarations may not need to be considered (as well as empty lines)
- After having a candidate VCC, checking if you should blame one of its ancestors may be beneficial.
- Semantic bugs (e.g. access control) seem to be a bit more difficult to trace.
- Concurrency bugs seem difficult to trace.
- Disregarding copied or refactored code may improve performance.
- If a function was replaced, it may be beneficial to look inside the function and see if the functionality changed.

- Not taking into account the comments may be beneficial.
- For the use-case of commit-based static analysis, noise in the training data may even improve performance. Noise in the evaluation data is not acceptable. Hence, the suggestion would be to use the Ubuntu ground truth as the testing dataset and improve the heuristic based on the recommendations in this section.
- Some vulnerability types (semantic, concurrency) are inherently more difficult to trace.

C.6.4 *Conclusion*

The VCCs from the Ubuntu Security Team seem to be as close to a ground truth as possible. They agree with the detailed manual analysis of the above-mentioned 10 bugs. There is no reason to believe that manual analysis by us would lead to better results than what is achieved by the Ubuntu Security Team (who takes in to account commit messages and RHL reports). Some vulnerabilities are inherently difficult to trace with automated heuristics. However, existing heuristics can be improved. Tracing the VCC manually seems much harder than implied in most previous work (maybe due to confirmation bias).

D

APPENDIX OF CHAPTER 5

D.1 SUMMARY OF DATA SOURCES

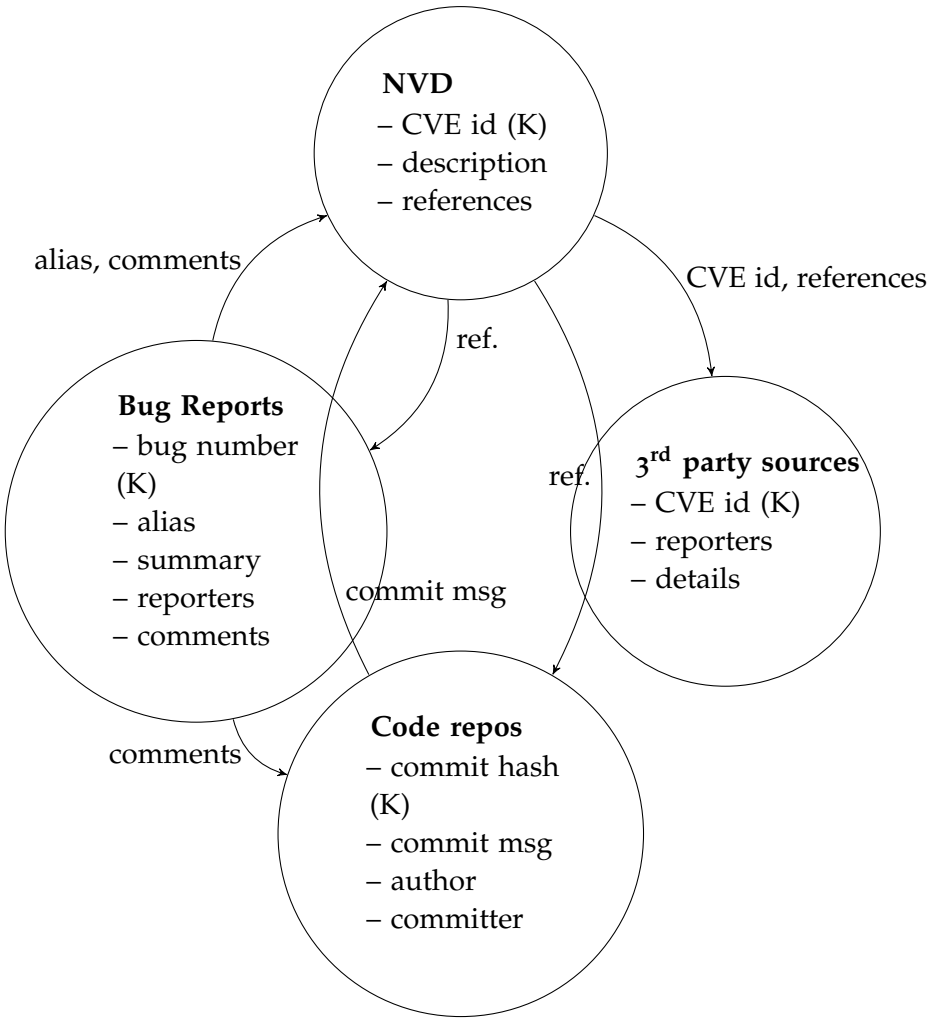


Figure D.1: Summary of collected data points and their connections. K stands for the primary (unique) key of the collection.

D.2 ADDITIONAL FIGURES

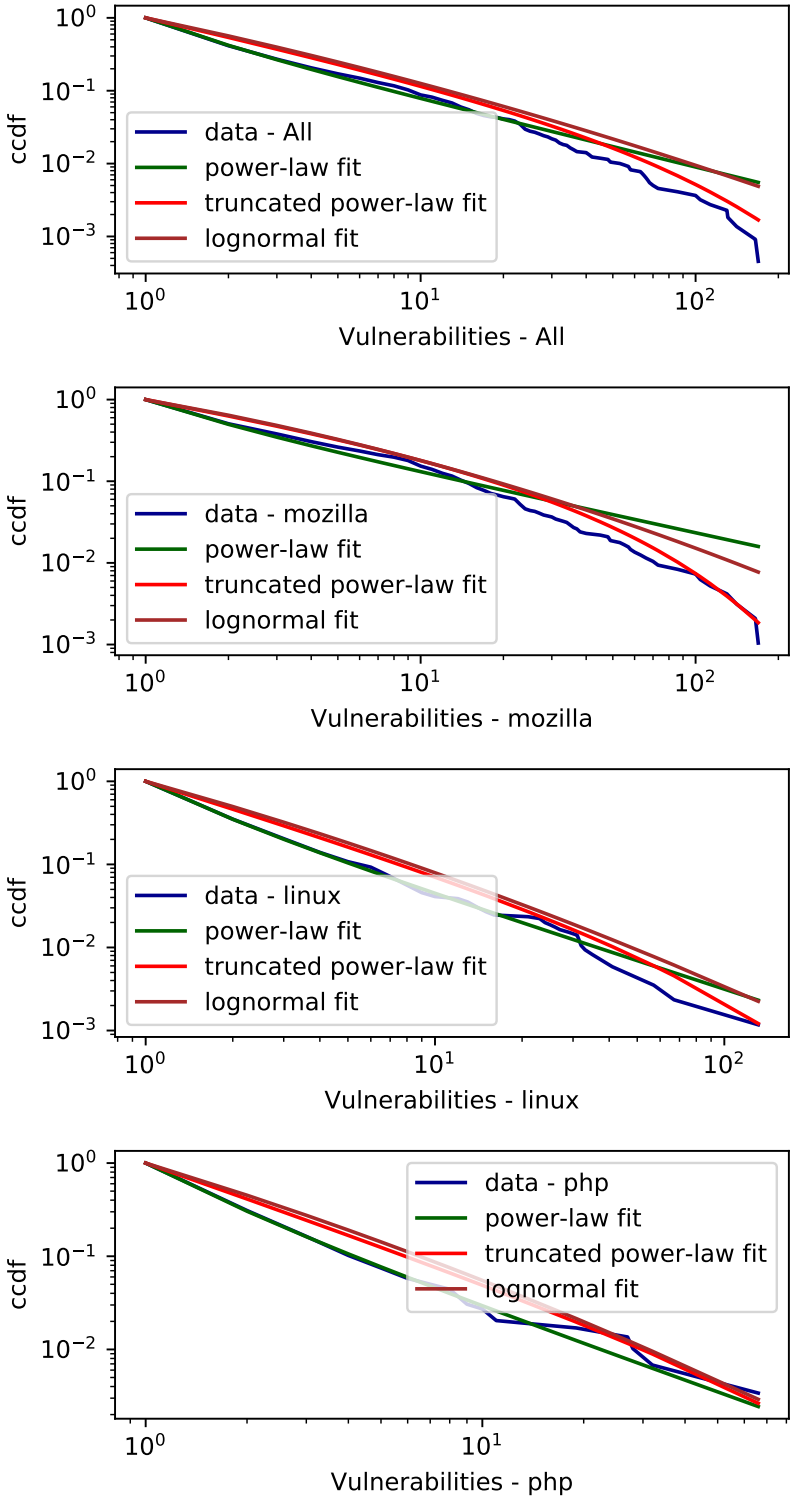


Figure D.2: Heavy-tailed distribution fits (complementary cumulative distribution function).

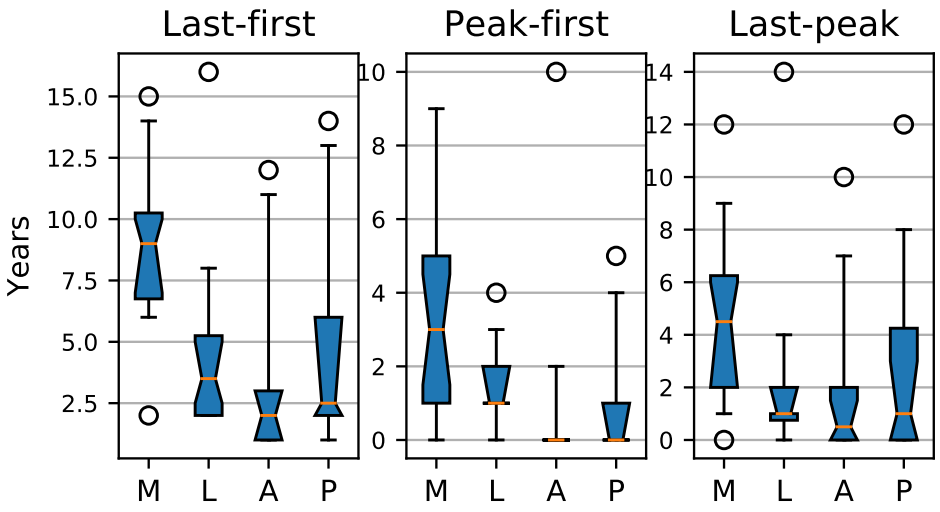


Figure D.3: For the top 20 (human) reporters for each project: time in years between (a) their first and more recent report until now (last), increased by one to measure period of engagement, (b) their first report and their first peak year, and (c) their first peak year and their last report until now ([5,95] whiskers). Letters in the x axis are the initials of the corresponding projects (Mozilla, Linux, Apache, PHP). 95% confidence intervals for the median, calculated via bootstrapping (10 000 times), are shown as notches.

BIBLIOGRAPHY

1. Ablon, L. & Bogart, A. *Zero Days, Thousands of Nights: The Life and Times of Zero-Day Vulnerabilities and Their Exploits* (RAND Corporation, Santa Monica, CA, 2017).
2. Alexopoulos, N., Brack, M., Wagner, J. P., Grube, T. & Mühlhäuser, M. *How long do vulnerabilities live in the code? A large-scale empirical measurement study on FOSS vulnerability lifetimes* in *31st USENIX Security Symposium, USENIX Security 2022 (to appear)* (USENIX Association, 2022).
3. Alexopoulos, N., Habib, S. M., Schulz, S. & Mühlhäuser, M. The Tip of the Iceberg: On the Merits of Finding Security Bugs. *ACM Trans. Priv. Secur.* **24** (2020).
4. Alexopoulos, N., Meneely, A., Arnouts, D. & Mühlhäuser, M. *Who are Vulnerability Reporters?: A Large-scale Empirical Study on FLOSS* in *ESEM '21: ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, Bari, Italy, October 11-15, 2021* (eds Lanubile, F., Kalinowski, M. & Baldassarre, M. T.) (ACM, 2021), 25:1–25:12.
5. Alhazmi, O. H. & Malaiya, Y. K. *Modeling the Vulnerability Discovery Process* in *16th International Symposium on Software Reliability Engineering (ISSRE 2005), 8-11 November 2005, Chicago, IL, USA* (IEEE Computer Society, 2005), 129–138.
6. Alhazmi, O. H. & Malaiya, Y. K. *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* (IEEE Computer Society, 2006), 343–352.
7. Alhazmi, O. H. & Malaiya, Y. K. *Quantitative vulnerability assessment of systems software* in *Annual Reliability and Maintainability Symposium, 2005. Proceedings.* (2005), 615–620.
8. Allodi, L. *Economic Factors of Vulnerability Trade and Exploitation* in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017* (eds Thuraisingham, B. M., Evans, D., Malkin, T. & Xu, D.) (ACM, 2017), 1483–1499.
9. Allodi, L., Massacci, F. & Williams, J. The Work-Averse Cyberattacker Model: Theory and Evidence from Two Million Attack Signatures. *Risk Analysis* (2021).
10. Alstott, J., Bullmore, E. & Plenz, D. powerlaw: a Python package for analysis of heavy-tailed distributions. *PloS one* **9,** e85777 (2014).

11. Amor, J. J., Robles, G., González-Barahona, J. M. & Rivas, F. Measuring Lenny: the size of Debian 5.0 (2009).

12. Anderson, R. *Security in open versus closed systems-The dance of Boltzmann, Coase and Moore* in *Conference on the Economics, Law and Policy of Open Source Software, Toulouse, France, 2002* (2002).

13. Arbaugh, W. A., Fithen, W. L. & McHugh, J. Windows of Vulnerability: A Case Study Analysis. *Computer* **33,** 52–59 (2000).

14. Bau, J., Bursztein, E., Gupta, D. & Mitchell, J. C. *State of the Art: Automated Black-Box Web Application Vulnerability Testing* in *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA* (IEEE Computer Society, 2010), 332–345.

15. Bilge, L. & Dumitras, T. *Before we knew it: an empirical study of zero-day attacks in the real world* in *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012* (eds Yu, T., Danezis, G. & Gligor, V. D.) (ACM, 2012), 833–844.

16. Bird, C., Gourley, A., Devanbu, P. T., Gertz, M. & Swaminathan, A. *Mining email social networks* in *Proceedings of the 2006 International Workshop on Mining Software Repositories, MSR 2006, Shanghai, China, May 22-23, 2006* (eds Diehl, S., Gall, H. C. & Hassan, A. E.) (ACM, 2006), 137–143.

17. Blyth, C. R. On Simpson's paradox and the sure-thing principle. *Journal of the American Statistical Association* **67,** 364–366 (1972).

18. Brack, M. *A Large-scale Statistical Analysis of Vulnerability Lifetimes in Open-Source Software. Bachelor thesis. Technical University of Darmstadt* 2020.

19. Brady, R. M., Anderson, R. J. & Ball, R. C. *Murphy's law, the fitness of evolving species, and the limits of software reliability* tech. rep. (University of Cambridge, Computer Laboratory, 1999).

20. Bugiel, S., Davi, L. V. & Schulz, S. *Scalable trust establishment with software reputation* in *Proceedings of the sixth ACM workshop on Scalable trusted computing, STC@CCS 2011, Chicago, Illinois, USA, October 17, 2011* (eds Chen, Y., Xu, S., Sadeghi, A. & Zhang, X.) (ACM, 2011), 15–24.

21. Canonical. *Ubuntu CVE Tracker* `https://git.launchpad.net/ubuntu-cve-tracker`. Accessed: 2020-03-18.

22. Christey, S. & Martin, B. *Buying into the bias: Why vulnerability statistics suck* 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`. 2013.

23. Clark, S., Frei, S., Blaze, M. & Smith, J. M. *Familiarity breeds contempt: the honeymoon effect and the role of legacy code in zero-day vulnerabilities* in *Twenty-Sixth Annual Computer Security Applications Conference, ACSAC 2010, Austin, Texas, USA, 6-10 December 2010* (eds Gates, C., Franz, M. & McDermott, J. P.) (ACM, 2010), 251–260.

24. Clauset, A., Shalizi, C. R. & Newman, M. E. J. Power-Law Distributions in Empirical Data. *SIAM Review* **51,** 661–703 (2009).

25. Cook, K. *Security bug lifetime* https://outflux.net/blog/archives/2016/10/18/security-bug-lifetime/. 2016.

26. Corbet, J. *Kernel vulnerabilities: old or new?* https://lwn.net/Articles/410606/. 2010.

27. Debian. *Debian Security Tracker* https://salsa.debian.org/security-tracker-team/security-tracker/-/tree/master/data/CVE.

28. *Debian security FAQ* , last accessed on 2017/05/08. 2016. https://www.debian.org/security/faq (2017).

29. Details, C. *Browse vulnerabilities by Date* https://www.cvedetails.com/browse-by-date.php. 2020.

30. Dulaunoy, A. *CVE-search* https://github.com/cve-search/cve-search. 2020.

31. Edwards, N. & Chen, L. *An historical examination of open source releases and their vulnerabilities* in *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012* (eds Yu, T., Danezis, G. & Gligor, V. D.) (ACM, 2012), 183–194.

32. Eusgeld, I., Freiling, F. & Reussner, R. Dependability metrics. *Lecture Notes in Computer Science. Springer-Verlag, Berlin, Germany* (2008).

33. Fang, M. & Hafiz, M. *Discovering buffer overflow vulnerabilities in the wild: an empirical study* in *2014 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '14, Torino, Italy, September 18-19, 2014* (eds Morisio, M., Dybå, T. & Torchiano, M.) (ACM, 2014), 23:1–23:10.

34. Favato, D., Ishitani, D., Oliveira, J. & Figueiredo, E. *Linus's Law: More Eyes Fewer Flaws in Open Source Projects* in *Proceedings of the XVIII Brazilian Symposium on Software Quality, SBQS 2019, Fortaleza, Brazil, October 28 - November 1, 2019* (eds Albuquerque, A. B. & de Paula Barros, A. L. B.) (ACM, 2019), 69–78.

35. Fenton, N. & Bieman, J. *Software metrics: a rigorous and practical approach* (CRC press, 2014).

36. Finifter, M., Akhawe, D. & Wagner, D. A. *An Empirical Study of Vulnerability Rewards Programs* in *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013* (2013), 273–288.

37. Frei, S. *Security econometrics: The dynamics of (in) security* Doctoral dissertation (ETH Zurich, 2009).

38. Frei, S., Schatzmann, D., Plattner, B. & Trammell, B. *Modelling the Security Ecosystem- The Dynamics of (In)Security* in *8th Annual Workshop on the Eco-*

nomics of Information Security, WEIS 2009, University College London, England, UK, June 24-25, 2009 (2009).

39. Goel, A. L. & Okumoto, K. Time-dependent error-detection rate model for software reliability and other performance measures. *IEEE transactions on Reliability* **28,** 206–211 (1979).

40. Hafiz, M. & Fang, M. Game of detections: how are security vulnerabilities discovered in the wild? *Empirical Software Engineering* **21,** 1920–1959 (2016).

41. Hardy, M. Pareto's law. *The Mathematical Intelligencer* **32,** 38–43 (2010).

42. Hata, H., Guo, M. & Babar, M. A. *Understanding the Heterogeneity of Contributors in Bug Bounty Programs* in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2017, Toronto, ON, Canada, November 9-10, 2017* (2017), 223–228.

43. Herley, C. & van Oorschot, P. C. *SoK: Science, Security and the Elusive Goal of Security as a Scientific Pursuit* in *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017* (IEEE Computer Society, 2017), 99–120.

44. *Is Software More Vulnerable Today?* enisa Cyber security info notes https://www.enisa.europa.eu/publications/info-notes/is-software-more-vulnerable-today. 2018.

45. Joh, H., Kim, J. & Malaiya, Y. K. *Vulnerability Discovery Modeling Using Weibull Distribution* in *19th International Symposium on Software Reliability Engineering (ISSRE 2008), 11-14 November 2008, Seattle/Redmond, WA, USA* (IEEE Computer Society, 2008), 299–300.

46. Joh, H. & Malaiya, Y. K. Periodicity in software vulnerability discovery, patching and exploitation. *Int. J. Inf. Sec.* **16,** 673–690 (2017).

47. Johnson, R. A., Wichern, D. W., *et al. Applied multivariate statistical analysis (Sixth Edition)* (Prentice hall Upper Saddle River, NJ, 2007).

48. Kellogg, M., Dort, V., Millstein, S. & Ernst, M. D. *Lightweight verification of array indexing* in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018* (eds Tip, F. & Bodden, E.) (ACM, 2018), 3–14.

49. Kim, J., Malaiya, Y. K. & Ray, I. *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* (IEEE Computer Society, 2007), 141–148.

50. Klein, G. *et al. seL4: formal verification of an OS kernel* in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009* (eds Matthews, J. N. & Anderson, T. E.) (ACM, 2009), 207–220.

51. Kotzias, P., Bilge, L., Vervier, P. & Caballero, J. *Mind Your Own Business: A Longitudinal Study of Threats and Vulnerabilities in Enterprises* in *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019* (The Internet Society, 2019).

52. Krebs, B. *A Time to Patch* http://voices.washingtonpost.com/securityfix/2006/01/a_time_to_patch.html. 2006.

53. Krebs, B. *Internet Explorer Unsafe for 284 Days in 2006* http://voices.washingtonpost.com/securityfix/2007/01/internet_explorer_unsafe_for_2.html. 2007.

54. Krebs, B. *Why Counting Flaws is Flawed* https://krebsonsecurity.com/2010/11/why-counting-flaws-is-flawed/. 2010.

55. Kunz, W. & Rittel, H. W. Issues as elements of information systems (Working Paper 131). *Center for Planning and Development Research, Berkeley, USA* (1970).

56. Lehman, M. M. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE* **68,** 1060–1076 (1980).

57. Lehman, M. M., Ramil, J. F., Wernick, P., Perry, D. E. & Turski, W. M. *Metrics and Laws of Software Evolution - The Nineties View* in *4th IEEE International Software Metrics Symposium (METRICS 1997), November 5-7, 1997, Albuquerque, NM, USA* (IEEE Computer Society, 1997), 20.

58. Li, F. & Paxson, V. *A Large-Scale Empirical Study of Security Patches* in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017* (eds Thuraisingham, B. M., Evans, D., Malkin, T. & Xu, D.) (ACM, 2017), 2201–2215.

59. Lotka, A. J. The frequency distribution of scientific productivity. *Journal of the Washington academy of sciences* **16,** 317–323 (1926).

60. Louridas, P., Spinellis, D. & Vlachos, V. Power laws in software. *ACM Trans. Softw. Eng. Methodol.* **18,** 2:1–2:26 (2008).

61. Luedtke, N. *linux kernel cves* https://github.com/nluedtke/linux_kernel_cves.

62. Maillart, T., Zhao, M., Grossklags, J. & Chuang, J. Given enough eyeballs, all bugs are shallow? Revisiting Eric Raymond with bug bounty programs. *J. Cybersecur.* **3,** 81–90 (2017).

63. Manadhata, P. K. & Wing, J. M. in *Moving Target Defense - Creating Asymmetric Uncertainty for Cyber Threats* (eds Jajodia, S., Ghosh, A. K., Swarup, V., Wang, C. & Wang, X. S.) 1–28 (Springer, 2011).

64. Manadhata, P. K. & Wing, J. M. An Attack Surface Metric. *IEEE Trans. Software Eng.* **37,** 371–386 (2011).

65. Massacci, F. & Nguyen, V. H. An Empirical Methodology to Evaluate Vulnerability Discovery Models. *IEEE Trans. Software Eng.* **40,** 1147–1162 (2014).

66. Mell, P., Scarfone, K. & Romanosky, S. *A complete guide to the common vulnerability scoring system version 2.0* in *Published by FIRST-forum of incident response and security teams* **1** (2007), 23.

67. Meneely, A., Srinivasan, H., Musa, A., Tejeda, A. R., Mokary, M. & Spates, B. *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* (2013), 65–74.

68. Meneely, A. & Williams, L. A. *Secure open source collaboration: an empirical study of linus' law* in *Proceedings of the 2009 ACM Conference on Computer and Communications Security, CCS 2009, Chicago, Illinois, USA, November 9-13, 2009* (eds Al-Shaer, E., Jha, S. & Keromytis, A. D.) (ACM, 2009), 453–462.

69. Meneely, A. & Williams, L. A. *Strengthening the empirical analysis of the relationship between Linus' Law and software security* in *Proceedings of the International Symposium on Empirical Software Engineering and Measurement, ESEM 2010, 16-17 September 2010, Bolzano/Bozen, Italy* (2010).

70. Meneely, A. *Vulnerability History Project* https://github.com/VulnerabilityHistoryProject. Accessed: 2020-03-18.

71. MITRE. *Common Weakness Enumeration* https://cwe.mitre.org/data/definitions/699.html. 2020.

72. Morgner, P., Mai, C., Koschate-Fischer, N., Freiling, F. C. & Benenson, Z. *Security Update Labels: Establishing Economic Incentives for Security Patching of IoT Consumer Products* in *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020* (IEEE, 2020), 429–446.

73. Munaiah, N. & Meneely, A. *Vulnerability Severity Scoring and Bounties: Why the Disconnect?* in *Proceedings of the 2nd International Workshop on Software Analytics* (Association for Computing Machinery, Seattle, WA, USA, 2016), 8–14.

74. Nappa, A., Johnson, R., Bilge, L., Caballero, J. & Dumitras, T. *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* (2015), 692–708.

75. Nayak, K., Marino, D., Efstathopoulos, P. & Dumitras, T. *Some Vulnerabilities Are Different Than Others - Studying Vulnerabilities and Attack Surfaces in the Wild* in *Research in Attacks, Intrusions and Defenses - 17th International Symposium, RAID 2014, Gothenburg, Sweden, September 17-19, 2014. Proceedings* (eds Stavrou, A., Bos, H. & Portokalidis, G.) **8688** (Springer, 2014), 426–446.

76. Nemec, M., Klinec, D., Svenda, P., Sekan, P. & Matyas, V. *Measuring Popularity of Cryptographic Libraries in Internet-Wide Scans* in *Proceedings of the 33rd Annual Computer Security Applications Conference, Orlando, FL, USA, December 4-8, 2017* (ACM, 2017), 162–175.

77. Nguyen, V. H., Dashevskyi, S. & Massacci, F. An automatic method for assessing the versions affected by a vulnerability. *Empir. Softw. Eng.* **21,** 2268–2297 (2016).

78. Nguyen, V. H. & Massacci, F. *An independent validation of vulnerability discovery models* in *7th ACM Symposium on Information, Compuer and Communications Security, ASIACCS '12, Seoul, Korea, May 2-4, 2012* (eds Youm, H. Y. & Won, Y.) (ACM, 2012), 6–7.

79. Nguyen, V. H. & Massacci, F. *The (un)reliability of NVD vulnerable versions data: an empirical experiment on Google Chrome vulnerabilities* in *8th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '13, Hangzhou, China - May 08 - 10, 2013* (eds Chen, K., Xie, Q., Qiu, W., Li, N. & Tzeng, W.) (ACM, 2013), 493–498.

80. Of Standards, U. N. I. & Technology. *National Vulnerability Database* https://nvd.nist.gov/home.

81. Ozment, A. *The Likelihood of Vulnerability Rediscovery and the Social Utility of Vulnerability Hunting* in *4th Annual Workshop on the Economics of Information Security, WEIS 2005, Harvard University, Cambridge, MA, USA, June 1-3, 2005* (2005).

82. Ozment, A. in *Quality of Protection - Security Measurements and Metrics* (eds Gollmann, D., Massacci, F. & Yautsiukhin, A.) 25–36 (Springer, 2006).

83. Ozment, A. *Improving vulnerability discovery models* in *Proceedings of the 3th ACM Workshop on Quality of Protection, QoP 2007, Alexandria, VA, USA, October 29, 2007* (eds Karjoth, G. & Stølen, K.) (ACM, 2007), 6–11.

84. Ozment, A. & Schechter, S. E. *Milk or Wine: Does Software Security Improve with Age?* in *Proceedings of the 15th USENIX Security Symposium, Vancouver, BC, Canada, July 31 - August 4, 2006* (ed Keromytis, A. D.) (USENIX Association, 2006).

85. Ozment, J. A. *Vulnerability discovery & software security* Doctoral dissertation (University of Cambridge, UK, 2007).

86. Pendleton, M., Garcia-Lebron, R., Cho, J. & Xu, S. A Survey on Systems Security Metrics. *ACM Comput. Surv.* **49,** 62:1–62:35 (2017).

87. Peng, H. & Payer, M. *USBFuzz: A Framework for Fuzzing USB Drivers by Device Emulation* in *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020* (eds Capkun, S. & Roesner, F.) (USENIX Association, 2020), 2559–2575.

88. Perl, H., Dechand, S., Smith, M., Arp, D., Yamaguchi, F., Rieck, K., Fahl, S. & Acar, Y. *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, Denver, CO, USA, October 12-16, 2015* (eds Ray, I., Li, N. & Kruegel, C.) (ACM, 2015), 426–437.

89. Pfleeger, C. P. The fundamentals of information security. *IEEE Software* **14,** 15–16 (1997).

90. Pfleeger, S. L. & Cunningham, R. K. Why Measuring Security Is Hard. *IEEE Secur. Priv.* **8,** 46–54 (2010).

91. Piantadosi, V., Scalabrino, S. & Oliveto, R. *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* (IEEE, 2019), 68–78.

92. Raymond, E. The cathedral and the bazaar. *Knowledge, Technology & Policy* **12,** 23–49 (1999).

93. Rescorla, E. *Security Holes . . . Who Cares?* in *Proceedings of the 12th USENIX Security Symposium, Washington, D.C., USA, August 4-8, 2003* (USENIX Association, 2003).

94. Rescorla, E. Is Finding Security Holes a Good Idea? *IEEE Secur. Priv.* **3.** (First presented at WEIS 2004), 14–19 (2005).

95. Roumani, Y., Nwankpa, J. K. & Roumani, Y. F. Time series modeling of vulnerabilities. *Comput. Secur.* **51,** 32–40 (2015).

96. Runeson, P. & Höst, M. Guidelines for conducting and reporting case study research in software engineering. *Empir. Softw. Eng.* **14,** 131–164 (2009).

97. Sabottke, C., Suciu, O. & Dumitras, T. *Vulnerability Disclosure in the Age of Social Media: Exploiting Twitter for Predicting Real-World Exploits* in *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015* (eds Jung, J. & Holz, T.) (USENIX Association, 2015), 1041–1056.

98. Sarabi, A., Zhu, Z., Xiao, C., Liu, M. & Dumitras, T. *Patch Me If You Can: A Study on the Effects of Individual User Behavior on the End-Host Vulnerability State* in *Passive and Active Measurement - 18th International Conference, PAM 2017, Sydney, NSW, Australia, March 30-31, 2017, Proceedings* (eds Kâafar, M. A., Uhlig, S. & Amann, J.) **10176** (Springer, 2017), 113–125.

99. Schechter, S. E. *Computer security strength and risk: a quantitative approach* Doctoral dissertation (Harvard University, 2004).

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

101. Serebryany, K., Bruening, D., Potapenko, A. & Vyukov, D. *AddressSanitizer: A Fast Address Sanity Checker* in *2012 USENIX Annual Technical Con-*

ference, Boston, MA, USA, June 13-15, 2012 (eds Heiser, G. & Hsieh, W. C.) (USENIX Association, 2012), 309–318.

102. Serebryany, K., Bruening, D., Potapenko, A. & Vyukov, D. *AddressSanitizer: A Fast Address Sanity Checker* in *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012* (eds Heiser, G. & Hsieh, W. C.) (USENIX Association, 2012), 309–318.

103. Shahzad, M., Shafiq, M. Z. & Liu, A. X. *A large scale exploratory analysis of software vulnerability life cycles* in *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland* (eds Glinz, M., Murphy, G. C. & Pezzè, M.) (IEEE Computer Society, 2012), 771–781.

104. Stallman, R. *et al.* Gnu general public license. *Free Software Foundation, Inc., Tech. Rep* (1991).

105. Stutzbach, D. & Rejaie, R. *Understanding churn in peer-to-peer networks* in *Proceedings of the 6th ACM SIGCOMM Internet Measurement Conference, IMC 2006, Rio de Janeriro, Brazil, October 25-27, 2006* (eds Almeida, J. M., Almeida, V. A. F. & Barford, P.) (ACM, 2006), 189–202.

106. Tan, L., Liu, C., Li, Z., Wang, X., Zhou, Y. & Zhai, C. Bug characteristics in open source software. *Empirical Software Engineering* **19,** 1665–1705 (2014).

107. Vahldiek-Oberwagner, A., Elnikety, E., Duarte, N. O., Sammler, M., Druschel, P. & Garg, D. *ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK)* in *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019* (eds Heninger, N. & Traynor, P.) (USENIX Association, 2019), 1221–1238.

108. Verendel, V. *Quantified security is a weak hypothesis: a critical survey of results and assumptions* in *Proceedings of the 2009 Workshop on New Security Paradigms, Oxford, United Kingdom, September 8-11, 2009* (eds Somayaji, A. & Ford, R.) (ACM, 2009), 37–50.

109. Voas, J., Ghosh, A., McGraw, G., Charron, F. & Miller, K. *Defining an adaptive software security metric from a dynamic software failure tolerance measure* in *Proceedings of 11th Annual Conference on Computer Assurance. COMPASS'96* (1996), 250–263.

110. Votipka, D., Stevens, R., Redmiles, E. M., Hu, J. & Mazurek, M. L. *Hackers vs. Testers: A Comparison of Software Vulnerability Discovery Processes* in *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA* (IEEE Computer Society, 2018), 374–391.

111. Wagner, C., Dulaunoy, A., Wagener, G. & Iklody, A. *MISP: The Design and Implementation of a Collaborative Threat Intelligence Sharing Platform* in *Proceedings of the 2016 ACM on Workshop on Information Sharing and Collaborative Security, WISCS 2016, Vienna, Austria, October 24 - 28, 2016* (eds Katzenbeisser, S., Weippl, E. R., Blass, E. & Kerschbaum, F.) (ACM, 2016), 49–56.

112. Wilk, M. B. & Gnanadesikan, R. Probability plotting methods for the analysis of data. *Biometrika* **55,** 1–17 (1968).

113. Willinger, W., Paxson, V. & Taqqu, M. S. Self-similarity and heavy tails: Structural modeling of network traffic. *A practical guide to heavy tails: statistical techniques and applications* **23,** 27–53 (1998).

114. Woo, S., Alhazmi, O. H. & Malaiya, Y. K. *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* (IEEE Computer Society, 2006), 103–110.

115. Xiao, C., Sarabi, A., Liu, Y., Li, B., Liu, M. & Dumitras, T. *From Patching Delays to Infection Symptoms: Using Risk Profiles for an Early Discovery of Vulnerabilities Exploited in the Wild* in *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018.* (2018), 903–918.

116. Yamaguchi, F., Golde, N., Arp, D. & Rieck, K. *Modeling and Discovering Vulnerabilities with Code Property Graphs* in *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014* (IEEE Computer Society, 2014), 590–604.

117. Yang, L., Li, X. & Yu, Y. *VulDigger: A Just-in-Time and Cost-Aware Tool for Digging Vulnerability-Contributing Changes* in *GLOBECOM 2017-2017 IEEE Global Communications Conference* (2017), 1–7.

118. Younis, A., Joh, H. & Malaiya, Y. *Modeling learningless vulnerability discovery using a folded distribution* in *Proc. of SAM* **11** (2011), 617–623.

119. Zhao, M., Grossklags, J. & Liu, P. *An Empirical Study of Web Vulnerability Discovery Ecosystems* in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015* (eds Ray, I., Li, N. & Kruegel, C.) (ACM, 2015), 1105–1117.

120. Zhu, X. & Böhme, M. *Regression Greybox Fuzzing* in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, CCS 2021 (to appear)* (2021).

DECLARATION

I hereby confirm that the submitted thesis with the title "New Approaches to Software Security Metrics and Measurements" has been done independently and without use of others than the indicated aids. I assure that I have not previously or concurrently applied for the opening of a promotion procedure with the doctoral thesis submitted here.

Darmstadt, 08.02.2022

Nikolaos Alexopoulos,
08.02.2022