# The Tip of the Iceberg: On the Merits of Finding Security Bugs

NIKOLAOS ALEXOPOULOS, Technische Universität Darmstadt, Germany
SHEIKH MAHBUB HABIB, Continental AG, Germany
STEFFEN SCHULZ, Intel Labs, Germany
MAX MÜHLHÄUSER, Technische Universität Darmstadt, Germany

In this paper, we investigate a fundamental question regarding software security: *Is the security of SW releases increasing over time?* We approach this question with a detailed analysis of the large body of open-source software packaged in the popular Debian GNU/Linux distribution. Contrary to common intuition, we find no clear evidence that the vulnerability rate of widely used software decreases over time: Even in popular and "stable" releases, the fixing of bugs does not seem to reduce the rate of newly identified vulnerabilities. The intuitive conclusion is worrisome: Commonly employed development and validation procedures do not seem to scale with the increase of features and complexity – they are only chopping pieces off the top of an iceberg of vulnerabilities.

To the best of our knowledge, this is the first investigation into the problem that studies a complete distribution of software, spanning multiple versions. Although we can not give a definitive answer, we show that several popular beliefs also cannot be confirmed given our dataset. We publish our *Debian Vulnerability Analysis Framework (DVAF)*, an automated dataset creation and analysis process, to enable reproduction and further analysis of our results. Overall, we hope our contributions to provide important insights into the vulnerability discovery process and help in identifying effective techniques for vulnerability analysis and prevention.

CCS Concepts: • **Security and privacy → Software and application security**.

Additional Key Words and Phrases: Empirical Study, Vulnerability Discovery Rate, Debian GNU/Linux

**ACM Reference Format:**
Nikolaos Alexopoulos, Sheikh Mahbub Habib, Steffen Schulz, and Max Mühlhäuser. 0000. The Tip of the Iceberg: On the Merits of Finding Security Bugs. *ACM Trans. Priv. Sec.* 00, 0, Article 000 ( 0000), 33 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 INTRODUCTION

Modern open-source software (OSS) systems comprise a multitude of interacting components, by different developers. Such software products are used in various critical aspects of our everyday lives, such as telecommunications, hospitals, transportations, etc., and therefore their security has become a critical issue. A number of high-profile vulnerabilities have gathered media attention over the last years. For example, *Shellshock* (CVE-2014-6271) was a vulnerability of the widely used

Authors' addresses: Nikolaos Alexopoulos, alexopoulos@tk.tu-darmstadt.de, Technische Universität Darmstadt, Germany; Sheikh Mahbub Habib, sheikh.mahbub.habib@continental-corporation.com, Continental AG, Germany; Steffen Schulz, steffen.schulz@intel.com, Intel Labs, Germany; Max Mühlhäuser, max@informatik.tu-darmstadt.de, Technische Universität Darmstadt, Germany.


2471-2566/0000/0-ART000 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

UNIX Bash shell that allowed attackers to gain complete control of a victim's machine without any prior knowledge of their credentials. Shortly after its disclosure, thousands of attacks took place that included compromising a number of machines and using them as a botnet to launch DDoS attacks [16]. Another infamous vulnerability, *Heartbleed* (CVE-2014-0160), was also discovered by white-hat hackers, but resulted in several exploits, like the leak of millions of hospital patient data [15]. The fact that known (discovered by white-hats) and patched vulnerabilities can cause such great disturbance, indicates that yet unknown vulnerabilities (zero-days), which can potentially affect billions of devices, are an even greater threat. At the same time, the amount of discovered security bugs is increasing at an impressive pace with over 20 thousand new vulnerabilities discovered through HackerOne's bug bounty program in 2016 alone [17], while the amount of CVEs reported in 2017 were more than double compared to any previous year, and a further increase followed in 2018.

The security community has come up with various defense mechanisms to prevent, locate and fix vulnerabilities, including *formal verification, static/dynamic code analysis* and *fuzzing*. Unfortunately, formal verification often requires a lot of manual extra work, and does not readily scale to large SW projects. Even widely used, high-risk components such as the *openssl* cryptographic library are not formally verified. In recent years, significant resources have therefore been allocated to automated vulnerability discovery. State of the art static analysis tools (e.g. [8, 10, 42]) offer ways to check software for possible vulnerabilities pre-release by pinpointing risky code segments. Additionally, there has been significant progress in the area of dynamic analysis and fuzzing tools (e.g. [29, 33, 36], as well as the AFL fuzzer[1]), that discover vulnerabilities via testing the runtime behaviour of the program. In practice, even high-criticality SW that was subject to extensive validation will often still contain (security) bugs, e.g. [9, 22].

There is a general feeling among security researchers and practitioners that the rise in the overall number of reported vulnerabilities in recent years is attributed to the sizeable increase of the codebase, and hence the increase of the overall attack surface, while the quality of established and widely used software components is improving. For instance, Ozment and Schlechter [27] report a gradual decrease in the number of foundational vulnerabilities of the OpenBSD operating system. In this paper, we set out to answer a fundamental question regarding the state of open-source software security, with the popular Debian GNU/Linux ecosystem as a case study: *Is the security of software increasing?* - Can we trust software more than in the past or are we introducing vulnerabilities at a faster rate than we are finding them? In the process of trying to find an answer, we developed the *Debian Vulnerability Analysis Framework*, a system for automatically collecting relevant data from publicly available sources, and for applying different data analysis techniques in a reproducible manner.

We choose the Debian distribution for our case study due to three important characteristics: (a) it is one of the biggest and most popular collections of OSS 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 the dataset is limited to Debian, it is likely that the results can be generalized to all general-purpose Linux distributions because the vast majority of SW projects and code base underlying the various distributions is identical: A vulnerability in a particular SW 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 is not currently

[1] http://lcamtuf.coredump.cx/afl/

known but in fact it is one of the motivations of our research. More information on how Debian works can be found in Section 2.

**Methodology and contributions:** We approach our central question in three steps, by forming three main hypotheses of increasing depth and detail, which we investigate in dedicated sections by analyzing plots and employing statistical tests to confirm the statistical significance of our observations. Our three main hypotheses and a short summary of our results is given in Table 1.

| |
|---|
| **– H1: The vulnerability rate of widely used OSS is showing signs of maturity (Section 4):** We found no clear signs of maturing behavior (e.g. 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. Generally, even experienced developer teams find it hard to produce secure software. |
| **– H2: There are less severe bugs and certain types are decreasing (Section 5):** H2.1 (severity): Although the ratio of high-severity vulnerabilities compared to the total is dropping, their absolute number does not show a sign of decrease. 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 methodologies targeting specific types do not have a measurable effect. |
| **– H3: Vulnerability prices for OSS in bug bounty programs are rising (Section 6):** Our investigation of a community-driven bug bounty program showed that there is no increase in the prices paid, even when considering only high severity vulnerabilities of popular OSS. Interestingly, the bounties paid on the platform overall, even when considering proprietary programs, have remained stable 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 prices did not increase). Our results are in contrast to recent reports of huge rewards offered for zero-days by offensive-oriented buyers. |

Table 1. Short summary of results and contributions

Regarding our main question, we conclude that there is no maturing effect evident for the security of OSS. 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[2] , 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.

[2]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.

Testing these hypotheses required analyzing data from Debian's security team's advisories to retrieve the vulnerability identifiers (id's for short) that affected the package versions of the stable distribution at each point in time. We repeated the procedure with the reports of the LTS (long-term support) team. Then, we retrieved the CVE numbers of those bugs, allowing us to better estimate when they were first publicly disclosed. We used this data to search for trends in the vulnerability rates of Debian overall, specific stable releases, and single programs. By utilizing the NVD's database, we then matched the CVEs with their assigned severity score (CVSS) and type (CWE), allowing us to make further inferences on trends considering bug severity and type (see H2). Finally, to investigate the hypotheses relating to bug bounty prices (H3), we retrieved all publicly available data from the HackerOne platform and analyzed it. In each case, the Laplace trend test and ordinary least squares regression (OLS) were used to support our observations from the manual analysis (plots) of the data.

To achieve our results, we developed the *DVAF* (*Debian Vulnerability Analysis Framework* – see Section 3) for vulnerability data analytics on software contained in Debian Linux. It automatically mines publicly accessible vulnerability repositories, such as the National Vulnerability Database (NVD) and Debian Security Advisories (DSA), in order to create datasets for applying different data analysis techniques in a reproducible manner. We publish the *DVAF* to enable further reproducible analysis (e.g. by investigating some of the questions we raise later in the paper).

## 2 BACKGROUND & TERMINOLOGY

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

**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 all widely used OSS in existence [5] (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 [35]. 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. The stable release is updated about every 2 years and only 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 the `testing` distribution. Debian releases are characterized by a number and a name, traditionally from Toy Story[3]. Security vulnerabilities are handled in a transparent manner by the Debian security team [1]. 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 trend 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 if 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 [28] (as it does not satisfy the independence requirement), it has been widely used by seminal previous work [27, 31, 43] in the area and therefore it is also employed by us (always with a pinch of salt). Although statistical tests are important to support our observations, they do not

[3]For an overview of Debian Releases see https://www.debian.org/releases/.

offer much to the presentation of the paper, and as such, graphical representations of the Laplace tests, as well as the complete statistical test results are confined to the appendix.

## 3 DATA COLLECTION AND THE DVAF

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. The framework consists of two basic components, data collection (`Input`) and data analysis (`Analysis`). These abstractions are in turn instantiated by various modules in an extensible way, as shown in Fig. 1. Our implementation consists of around 1 000 lines of python3 code and is available as open-source on github[4].

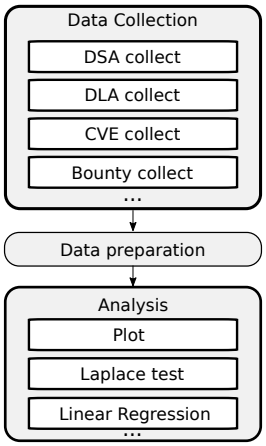



Fig. 1. The DVAF's extendable architecture and workflow

The currently implemented modules work as follows:
**DSA Input:** 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 Input:** 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.
**CVE Input:** CVE data, including the date of the CVE, and various metadata (CVSS score, CWE type etc.) are collected by employing the `cve-search`[5] tool, a tool for local lookups on reported CVEs.
**Bug bounties Input:** 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.)

[4]https://github.com/nikalexo/DVAF
[5]https://github.com/cve-search/cve-search

by utilizing the platform's API[6].

**Data preparation:** The preparation of the data consists of manual corrections of known mistakes in the vulnerability reports, 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 vulnerability 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, however this can be configured. All data points are stored in json text representations of python dictionaries. Vulnerabilities are grouped in month intervals and a python dictionary `src2month` holds the time-series corresponding to each source package.

**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.

## 4 VULNERABILITIES IN DEBIAN

In this section, we present an overview of the Debian ecosystem w.r.t. its security characteristics and draw some interesting conclusions, aiming to investigate H1. Contrary to previous studies (e.g. [2, 12]), 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. The question we ask ourselves in this section is whether there are signs that the security quality of software 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 slowing down. Therefore, we choose to study the raw vulnerability numbers discovered in Debian packages, rather than their vulnerability density, a choice that we also discuss later on. In Fig. 2, we see the distribution of vulnerabilities among source packages of Debian for the years 2001-2018[7]. 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 the appendix).

**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

[6]During the revision process of the paper, 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.

[7]Our analysis in this paper is limited to data until the end of 2018 for two reasons. First, it is the last completed year at the time of writing. Second, some time is needed for entries in the NVD to be filled with information (e.g. severity, type) regarding discovered vulnerabilities.

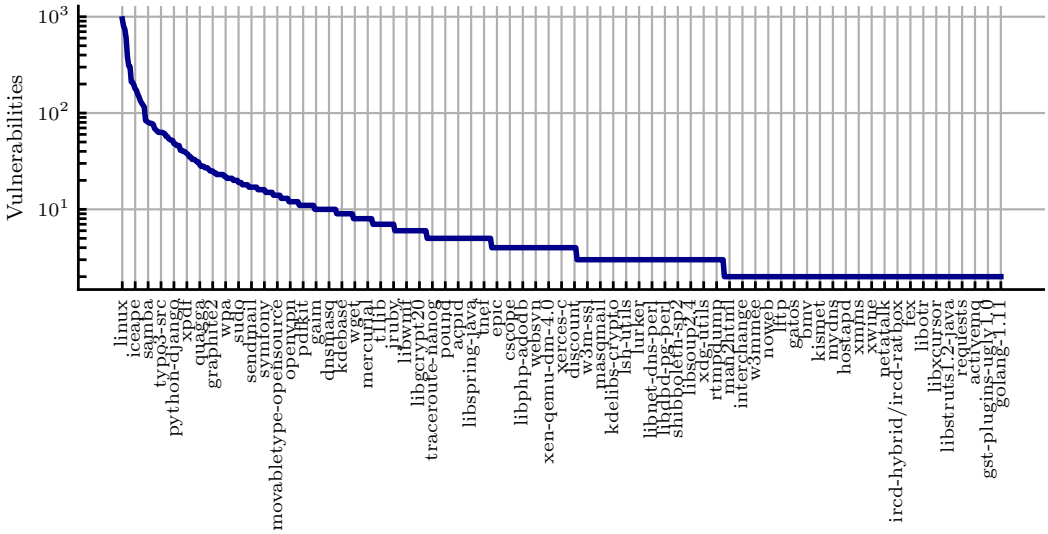
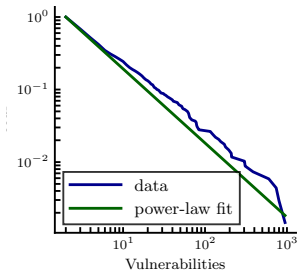
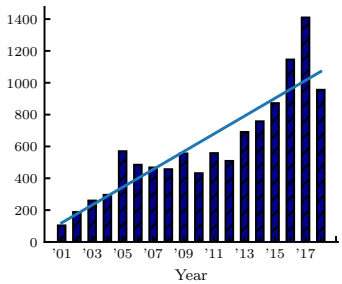


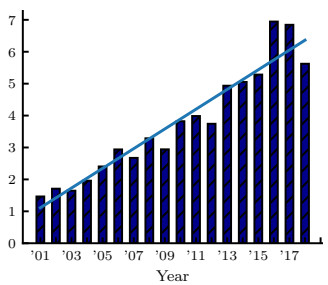

Fig. 2. The distribution of vulnerabilities per package (years 2001-2018). Every tenth package name appears on the x axis for space reasons. The y axis is logarithmic. Packages with at least two vulnerabilities are taken into account.




(a) A log-log plot (complementary cumulative distribution function) of the distribution of Fig. 2.




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




(c) Average vulnerabilities per package (that had at least one security incident in that year) per calendar year (2001-2018).

Fig. 3. Vulnerabilities: distribution and trends.

(Fig. 3a) of the probability distribution of the data in (double) logarithmic axes, we can observe a near-straight line, indicative of a potential power-law distribution (Zipfian). Power-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. [13] and the `powerlaw` package [4], 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

and $p$ their probability density function[8]. 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, identifying the cause of this distribution is not trivial and is required in any case, although it falls out of the scope of this study. Other heavy-tailed distributions (power-law with exponential cut-off, lognormal) are also possible fits, and in general very difficult to statistically disprove [13].

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 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. 17 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 goes by is required for more definitive statistical tests.

**Global observations:** Table 2 presents the 20 top vulnerable Debian source packages of all time. 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). The Linux kernel turns out to be the most vulnerable component, followed by the two main browsers in use (Chromium[9] and Firefox ). The total number of unique vulnerabilities[10] reported in the 18 year period was 10 716, with the kernel accounting for around 9 % of the total. 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 compared to the next most vulnerable package (the Linux kernel) which was affected by 160 (around 7 % of the total).

Concerning the total number of vulnerabilities reported in the Debian ecosystem w.r.t. time, Fig. 3b shows a clear upward trend as the years go by. Can this mean that the security quality of the software 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 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 maintainers, 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 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

[8]It is common in literature (e.g. [40]) 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 good enough and explained most of our data points.

[9]open-source code-base of the proprietary Google Chrome browser.

[10]Note that a vulnerability may affect more than one packages. More discussion on this follows in this section.

| package | # total | rank | $\#_{17-18}$ | $\text{rank}_{17-18}$ |
|---|---|---|---|---|
| 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 2. The top twenty packages with the most vulnerabilities in time periods (i) 2001-2018 and (ii) 2017-2018.

in Fig. 3c. In the latter figure we can even see a smoother, clearer upward trend compared to Fig. 3b, although the slope of the trend is nearly identical. 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 2, the trend 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, which 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.

PHP: is a popular server-side scripting language that is used by 79% of all websites whose server-side programming language is known[11]. We will look into the transition from php5 to the next

[11]https://w3techs.com/technologies/details/pl-php (November 2019)

version php7 [12] (php6 never made it to the public). The vulnerability history of php5 (see Fig. 4) 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 amount of vulnerabilities after this point

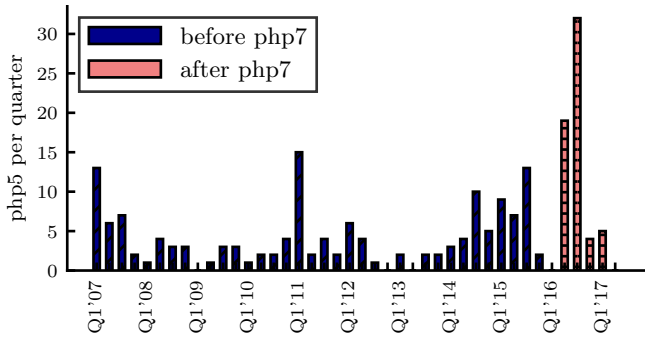



Fig. 4. Vulnerabilities of php5, during its presence in the stable release, before and after the introduction of the next version (php7) in testing. Vulnerability rate: (a) before the launch of the new version: ≈ 4 vuln./quarter; (b) after the launch of the new version: ≈ 10 vuln./quarter.

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 manual inspection of all security incidents tracked by the Debian Security Bug Tracker[13], 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[14]. 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 seems not to 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.

OpenJDK: We repeat the experiment with OpenJDK, 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. 5, we see the vulnerabilities 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.

[12] Official package name php7.0

[13] https://security-tracker.debian.org/tracker/

[14] Attribution of vulnerabilities to affected versions was made according to information about patched versions in the Debian Security Bug Tracker.

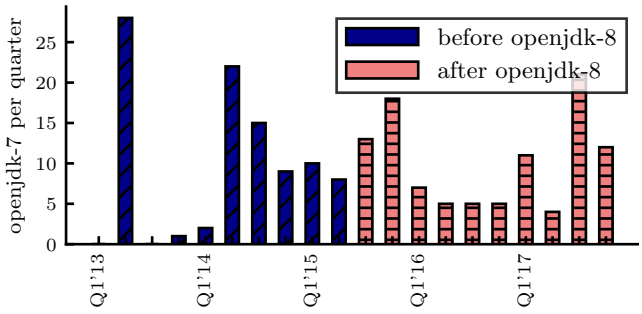



Fig. 5. Vulnerabilities of openjdk-7, during its presence in the stable release, before and after the introduction of the next version (openjdk-8) in testing. Vulnerability rate: (a) before the launch of the new version: ≈ 11.3 vuln./quarter; (b) after the launch of the new version: ≈ 10.6 vuln./quarter.

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. 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[15] team (May 2018). In Fig. 6, we see the distribution of vulnerabilities per quarter, starting from the release of Wheezy. Even for a

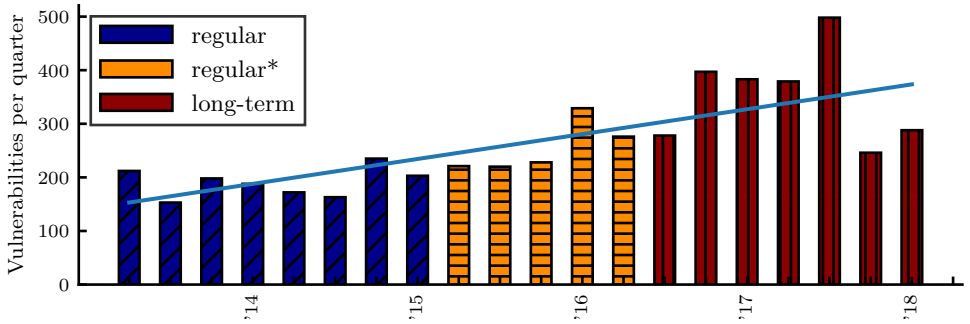



Fig. 6. Vulnerabilities that affected packages of the Wheezy Debian release.
*From Q2/2015 to Q2/2016, both Debian 7 (Wheezy) and 8 (Jessie) were supported by the regular security team. This was due to the fact that current Debian practice is that when a new stable version is released, the previous one (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.

specific stable release of Debian, we can observe a clear upward trend that continues in the LTS phase of the software (as expected the trend is statistically significant). 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.

**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

[15]Starting from 2014, Debian includes an LTS program, in order to extend support for any release to at least 5 years in total.

attack vector [25], 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. 7, we see the number of vulnerabilities that affected at least two packages over time, and in Table 3 the most prevalent package sets jointly affected by vulnerabilities. We observe that

| 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. Most common sets of packages jointly affected by vulnerabilities.

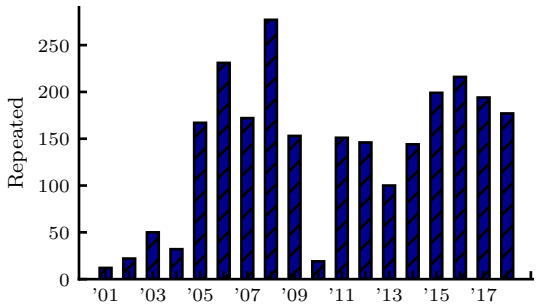



Fig. 7. Vulnerabilities affecting at least two Debian packages.

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*[16]), and mariadb starting as a fork of mysql. Although for all results presented in this paper we count each vulnerability only once even though it may affect one or more packages, we did not see any qualitative differences when counting vulnerabilities multiple times in any of the trends we investigated. Therefore, we can safely conclude that the shared code effect does not significantly affect the overall trends of vulnerability discoveries in Debian.

**Discussion:** In this section, we investigated the distribution of vulnerabilities in the Debian ecosystem. After detailed examination of the vulnerabilities reported for both individual widely used packages (case studies on PHP and OpenJDK), and a specific stable release of Debian, we conclude that the number of vulnerabilities does not visibly decrease over time, even for software that has been stable for many years. To the contrary, we discerned a relatively stable rate of vulnerabilities, that shows signs of statistically significant increase over time. In other words, we are still in the phase of *the more we look - the more we find*. Although automated security tools and manual security inspection are becoming more widespread and effective, we have not reached the point of curbing the vulnerability rate yet.

Our results draw interesting comparisons to studies performed over a decade ago. Rescorla claimed in [31] 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 [27] tried to challenge Rescorla's claims and found evidence of a decrease in the vulnerability rate of the foundational vulnerabilities of OpenBSD in a 7.5 year interval (statistically significant decrease was observed after 5 years from

[16]https://wiki.mozilla.org/Core

the release of the sotware). This can be attributed to the very few features available in OpenBSD and maybe even to its relatively low popularity. 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 more difficult to find, or we introduce vulnerabilities faster than we can find them.

Our results can also be compared to the more recent ones of Edwards and Chen [14] 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 the initial 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 study, it increased again when the new version entered the testing phase. 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. Interestingly, Clark et al. [12] 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 is subjective, and may vary considerably among different researchers/testers. Conclusions made in [24] 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 automated tools do not seem to have a measurable impact.

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 [20] and revised in the 1990's [21], with the addition of, among other, the Law of Declining Quality. This law states that "The quality of E-type systems[17] 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.

## 5 VULNERABILITY SEVERITY AND TYPES

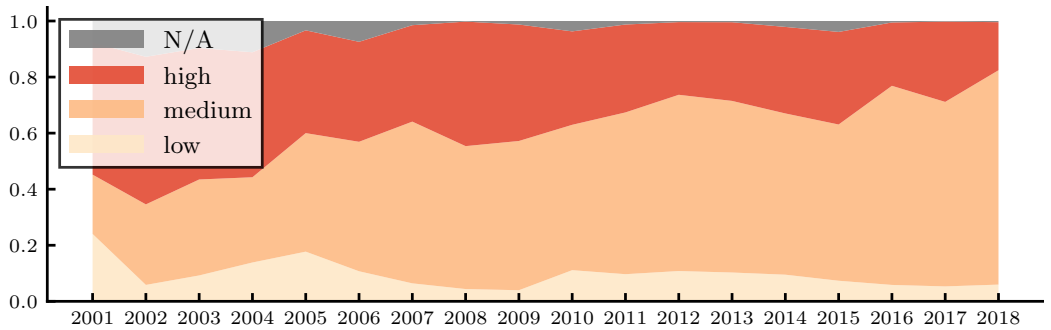



Fig. 8. Vulnerabilities severity of the stable release over time.

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

After establishing a general picture of the vulnerability landscape, we move on to investigate our hypotheses regarding vulnerability severity and type.

**H2.1 CVE Severity:** Since we established that there is no observable decrease in the overall vulnerability rate of OSS, we proceed to investigate the theory that although more bugs are discovered, they are less critical and more difficult to exploit than before. Then, one would expect a decrease of the ratio of high-severity vulnerabilities, compared to less critical ones. In Fig. 8, we see the progression of the ratio of low, medium, and high severity vulnerabilities, as classified by their CVSS score. An obvious trend 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 critical vulnerabilities shows a decrease recently, but are high severity vulnerabilities becoming rarer?

To test this hypothesis, Fig. 9 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

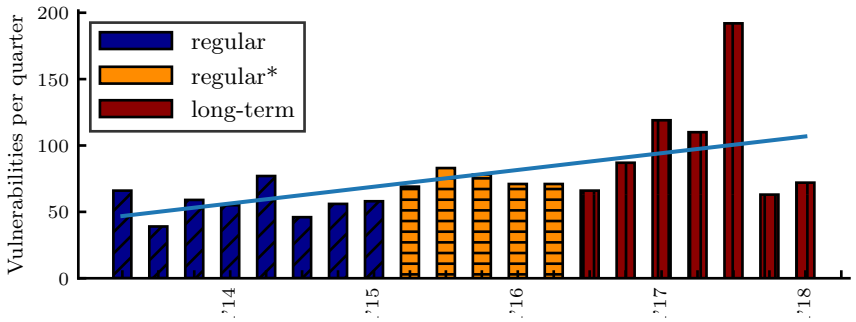



Fig. 9. High severity vulnerabilities of Debian Wheezy. The irregular peak of Q3'17 can be largely attributed to DLA 1097-1 which contained 86 CVEs affecting tcpdump.

to the contrary a statistically 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 such comparisons are consciously left as future work.

**H2.2 CVE 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 4.

| root | 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 4. Vulnerability type classification per root CWE number with most dominant examples in our dataset.

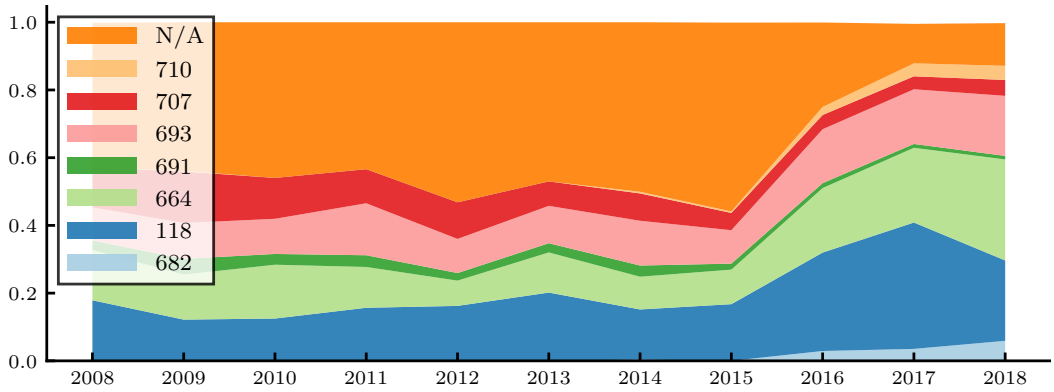



Fig. 10. Vulnerability types per year of Debian stable. Labels correspond to root CWE numbers (research view).

The progression of bug types over time is shown in Fig. 10. 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 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 neutralization 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

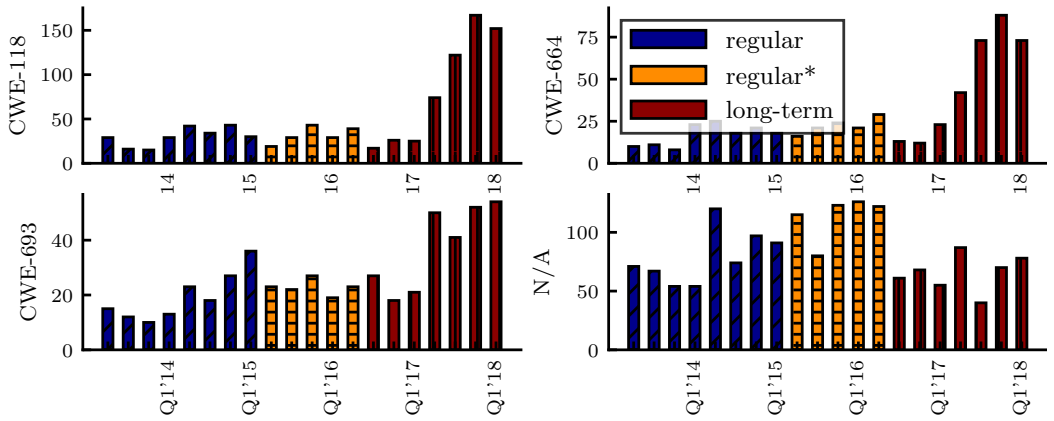



Fig. 11. Main vulnerability types of Debian Wheezy, including LTS.

type, which is reasonable as these vulnerabilities are suitable for automatic detection. In fact, this result supports the claims of [6], 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. 11, 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 seems 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:** 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 [23] vulnerability lifetime 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[18] 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

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

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. It will be interesting to see if such tools have a measurable impact in the near future. Future work on measuring the effectiveness of automated tools in practice would be appreciated by the community.

It is also valuable to compare our observations to the ones of Tan et al. [37] 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 [37] there is no trend analysis of security bug types over time to compare with our findings.

## 6 BUG BOUNTY PROGRAMS

So far, the security quality of OSS does not show clear signs of improvement. Another popular argument, supported by numerous media reports, is that vulnerabilities are getting more and more expensive and bug bounties are increasing. This would indicate that they are becoming harder to find and software is indeed becoming more secure. In this section, we investigate this hypothesis (H3) and what it tells us about the security quality of OSS in particular, by looking into the publicly visible reports of the well-known Bug Bounty platform HackerOne[19].

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. [24, 43]). The Internet Bug Bounty (IBB) program[20] is a community-driven initiative to award rewards for bugs affecting important OSS[21]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.

Table 5 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). 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". A total of 569 reports have been awarded a 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 paper 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 OSS 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. 12 presents the number of reports that were awarded bounties by the IBB from 2014 until November 2018. Notably, the number of bounties paid shows

[19]https://www.hackerone.com/
[20]https://internetbugbounty.org/
[21]with the notable exception of Adobe Flash until 2016.

| Component | Bounty # | Disclosed # | Total amount ($) | Max amount ($) | Average amount ($) |
|---|---|---|---|---|---|
| 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 5. IBB dataset summary snapshot on November 2018.

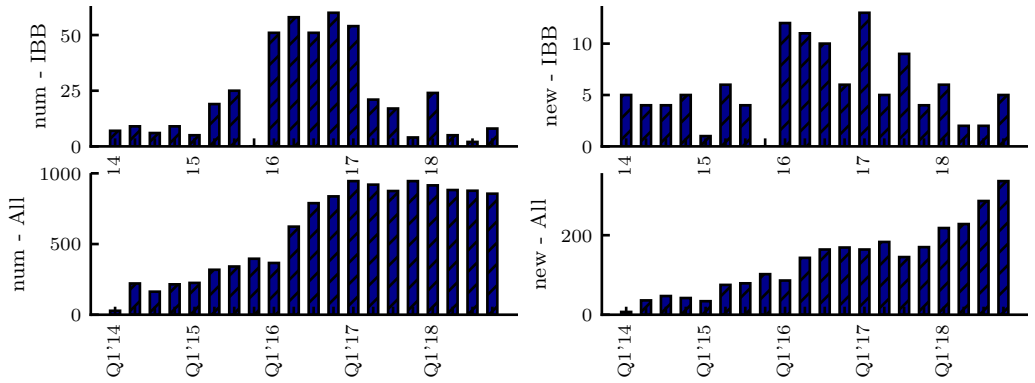



Fig. 12. Number of claimed bounty reports (left) and new reporters (right) entering the program for IBB (top row) and the HackerOne platform overall (bottom row) over time.

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 if reports, and is more similar to the traditional reliability-style hardening behavior we expected to see (but did not) for vulnerability discoveries in Debian. Is this a contradiction to our earlier observations? An indicator to support this hypothesis is that the monetary amount of the rewarded bounties increases over time. To test it, we look into the progression of awarded bounties over time.

Fig. 13 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. 13 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. 12 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 other programs that have entered the platform, since bugs outside the platform continue getting reported at a non-negligible rate.

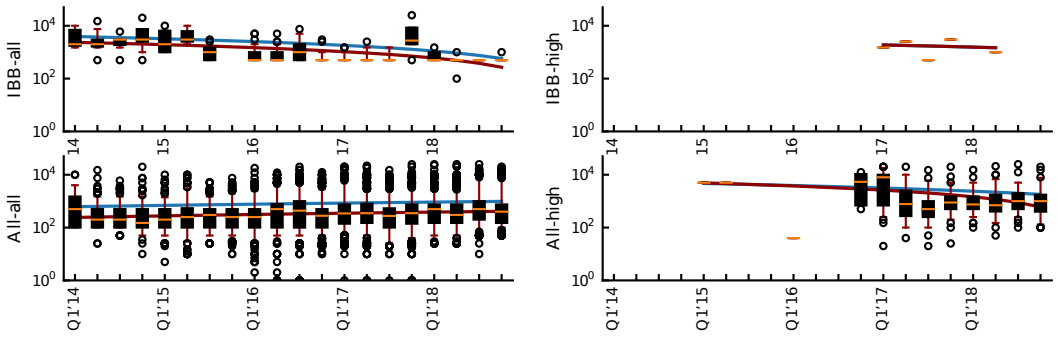



Fig. 13. (5-95%) box plot of USD paid over time for all (left) and only high/critical (right) severity vulnerabilities. Trend lines for the average (blue) and median (dark red). The only significant OLS trend comes from the top left plot concerning all the bugs reported in the IBB: a statistically significant decrease of the average as well as the median bounty. Detailed statistical test results in the additional material.

Upon closer re-inspection of Fig. 12, 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 [26] (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.

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. 14, 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

[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

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 (old ones, as well as new) may not have expended a large portion of their effort on the IBB after 2017, instead engaging in the program rather superficially.

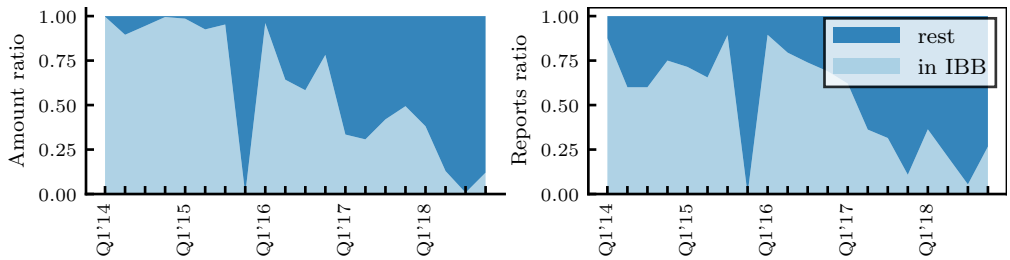



Fig. 14. Ratio of bounty amounts (left) and number of reports (right) of IBB reporters (at least 1 IBB report at some point in time) comparing reports in the IBB program against reports for other programs in HackerOne over time.

**Discussion:** It is valuable to compare our results with some recent important papers in the area. Zhao et al. [43] (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. [24] 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 [24, 43]. Allodi's study of an underground black marketplace [3] 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 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 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 OSS community may be considered altruistic and/or content with "swag"/reputation rewards for discovering vulnerabilities, and therefore the attractiveness of OSS bounty programs cannot be purely evaluated by the monetary reward. 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.

[25]https://www.zerodium.com/

## 7 THREATS TO VALIDITY

In this paper, 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. There are a number of factors that may influence our results:

1. (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 security quality of the stable versions of software is increasing, or we are introducing vulnerabilities faster than we find them. 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 observations that most vulnerabilities of PHP and OpenJDK are located in core components, an observed drop in the vulnerability density rate would convey a false feeling of increasing quality.

2. (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 SW security are the speed, consistency and reach of patch distribution. In this work, 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 SW 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?

3. (Security architectures) Modern SW protection and isolation technology can prevent vulnerabilities from interfering with a particular platform or transaction, or or make it hard to weaponize or scale a SW 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 and use-cases, or in combination with other vulnerabilities. In that sense, our dataset reflects the effects of inherent SW protections that objectively affect or prevent a potential SW 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 SW vulnerabilities and not a particular platform/use-case, we believe this is a fair representation of deployed mitigation strategies.

5. (Data quality and availability) As evident by our observations in Section 5, 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. 11, erring on the side of caution. Fine-grained analysis of type trends should be performed in the near future. Additionally, the NVD is known to contain inaccuracies regarding the vulnerable program versions (e.g. as documented in Nappa et al. [25]). 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 judgement. 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. 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.

Furthermore, 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 OSS on HackerOne, in comparison to big companies, e.g. Google, needs to be further investigated.

6. ("You did not test X") There are many more aspects and hypotheses that could be investigated. We encourage researchers to validate our results and explore further ideas using our published framework *DVAF*.

## 8 RELATED WORK

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 [18] and patching process [23] (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, we will focus on works about security issues.

Our work is motivated by the results of Ozment and Schechter [27], presented more than a decade ago. In the paper, the authors look for evidence that the quality of software is increasing, by examining the vulnerability rate of the OpenBSD operating system. They conclude that the rate of *foundational* vulnerabilities, i.e. vulnerabilities that were part of the first stable release of the product, is decreasing with time, and present this as an argument that the security of the software is increasing. In our investigation, we concluded that this is not the case for a large fraction of the software (in Debian), when considering security over multiple *stable* versions of the component. Our results support the line of thought of the 2005 paper by Rescorla [31], who reported no measurable evidence of an improvement in the security of software, and proposed that we should perhaps spend our time otherwise. We show that more than a decade later, Rescorla's concerns still apply (see also Sec. 4). 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 [27], however looking only into foundational vulnerabilities would not make much sense since the amount of code of the first stable release of many OSS projects that is also part of the current stable release is minimal (e.g. for the Linux kernel).

Apart from the ones mentioned above, there is a lot of work 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. In [11], the authors use 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 also follow a similar methodology to collect a part of our data, but our analysis has a very different goal and detail level. In [12], the authors study the effect of code familiarity in the vulnerability discovery process. They find 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 is complementary to theirs, as we study the subsequent stages of the vulnerability discovery process, from the moment a component becomes part of a stable distribution. In [2], time-based and effort-based models are shown to be 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 investigate a much larger body of OSS 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 with this paper. In [19], the authors try 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. Their results are in line with our analysis of OpenJDK and PHP, as they also 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 the user base.

The line of work based on the WINE dataset [7, 25, 41] (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. [25] study 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 is complementary to our research, as it investigates a latter part of the vulnerability lifecycle (i.e. how fast users install patches after they have been made available), while we are investigating 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 motivated us to investigate the issue in Section 4.

Li and Paxson [23] perform a large-scale study on security patches based on the NVD and commits in the projects' version control systems. They focus 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), while we investigate how characteristics evolve over time, something not addressed by them. Thus, unfortunately we cannot directly compare our results to theirs. We discuss our findings regarding vulnerability types in relation to theirs in Section 5. Overall, our findings nicely complement and support their claim for the need for more effective testing and auditing processes for OSS. Not only do vulnerabilities remain in the code for a long period of time (as they find), but their discovery rate seems not to decline even when considering only severe vulnerabilities or specific types (as we find). We repeat their concerns regarding the need for more effective vulnerability-finding tools and development processes in Section 9. In the same section, we also provide new important insights gained from our results: the need for longer-term support, more effective bug bounty programs for OSS, and new more expressive security metrics and continuous measurement.

In [32], the authors fit several time-series models to the vulnerability rate of a number of software components, and report reasonable prediction accuracy. Although we do not attempt any predictions, this could be an interesting future work direction, as we have a richer and more complete dataset. In [14], a decrease in the vulnerability rate of specific series of software releases is noted after 3 to 5 years from the initial release. However, as we have noted, the release of the next series may again contribute to an increase of the vulnerability rate. Finally, in [37], the authors study bugs in three open source projects, the Linux kernel, Mozilla and Apache. Their study studies 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 have compared our results to theirs in Section 5. Since generic bugs and vulnerabilities behave very differently, our work is complementary to theirs, with minimal overlap. Furthermore, we consider the whole Debian ecosystem, while they only consider a subset of it consisting of three components.

## 9 DISCUSSION AND RECOMMENDATIONS

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 [30], 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 [7], it is valuable for organizations to share information regarding attacks with each other. Therefore, effective information sharing platforms are required. Efforts such as MISP [39] are encouraging and should be pursued further.

**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 syzcaller[26] paired with automatic continuous fuzzing of kernel branches (syzbot) are steps in the right direction. Furthermore, Google's recently 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 [34]), as well as detectors for other dangerous behavior (e.g. UBSan[28] for undefined behavior), are needed.

**Need for more attractive bug-bounty programs for OSS:** Our results showcase that bug bounty programs can be very 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 OSS projects. The community should look to increase the attractiveness of OSS 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.

**Need for more expressive security metrics and continuous measurement:** In this paper, 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 express the security quality of software. To the best of our knowledge this is an open problem. Furthermore, studies investigating the life span of

[26] https://github.com/google/syzkaller
[27] https://google.github.io/oss-fuzz/
[28] https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html

vulnerabilities (e.g. [23, 27]) 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. More work on this problem is needed. 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.

**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 [38] 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. The same can be said for programming languages designed for safety, such as Rust[33], although the vast amount of systems code that is written in traditional programming languages that are not designed with safety in mind (e.g. C/C++), means that this later countermeasure is more long-term oriented than the other mitigations mentioned above. Improvements of all the above-mentioned countermeasures, as well as development of new ones targeting other types of exploits should be emphasized imminently.

## 10 CONCLUSION

In this paper, we presented the results of our large-scale investigation into the Debian GNU/Linux vulnerability landscape, from a different perspective compared to most of the previous work on software security. As the title of the paper suggests, we arrived at the conclusion that there is no compelling evidence suggesting that OSS products are becoming more secure. Overall, especially when considering multiple stable versions, we conclude that we are not finding vulnerabilities fast enough, and therefore we are not making the iceberg any smaller. We further supported this result with an investigation of vulnerability trends for notable bug types, as well as for trends in bug bounty programs, showing that the effect of automatic analyzers and tools seems rather limited, while bug bounty programs for OSS lack in long-term attractiveness. Our results raise questions about the effectiveness of applied processes, and along with the *DVAF*, open the door for further systematic and scientific quantitative studies.

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, while vulnerability reporting practices are showing signs of improvement. That being said, the need for better metrics and measurement practices (both micro- and macroscopic), as well as studies going further than the measurement of the vulnerability discovery rate, are at an all-time high.

## REFERENCES

[1] 2016. Debian security FAQ. https://www.debian.org/security/faq
[2] Omar H Alhazmi and Yashwant K Malaiya. 2005. Quantitative vulnerability assessment of systems software. In *Reliability and Maintainability Symposium, 2005. Proceedings. Annual.* IEEE, 615–620.
[3] Luca Allodi. 2017. Economic factors of vulnerability trade and exploitation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security.* ACM, 1483–1499.

[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/

[4] Jeff Alstott, Ed Bullmore, and Dietmar Plenz. 2014. powerlaw: a Python package for analysis of heavy-tailed distributions. *PloS one* 9, 1 (2014), e85777.

[5] Juan José Amor, Gregorio Robles, Jesus M González-Barahona, and Francisco Rivas. 2009. Measuring Lenny: the size of Debian 5.0. (2009).

[6] Jason Bau, Elie Bursztein, Divij Gupta, and John Mitchell. 2010. State of the art: Automated black-box web application vulnerability testing. In *2010 IEEE Symposium on Security and Privacy*. IEEE, 332–345.

[7] Leyla Bilge and Tudor Dumitras. 2012. Before we knew it: an empirical study of zero-day attacks in the real world. In *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 833–844.

[8] Priyam Biswas, Alessandro Di Federico, Scott A. Carr, Prabhu Rajasekaran, Stijn Volckaert, Yeoul Na, Michael Franz, and Mathias Payer. 2017. Venerable Variadic Vulnerabilities Vanquished. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 186–198.

[9] Matteo Bortolozzo, Matteo Centenaro, Riccardo Focardi, and Graham Steel. 2010. Attacking and fixing PKCS#11 security tokens. In *Conference on Computer and Communications Security (CCS)*. 260–269.

[10] Fraser Brown, Shravan Narayan, Riad S Wahby, Dawson Engler, Ranjit Jhala, and Deian Stefan. 2017. Finding and Preventing Bugs in JavaScript Bindings. In *Security and Privacy (SP), 2017 IEEE Symposium on*. IEEE, 559–578.

[11] Sven Bugiel, Lucas Vincenco Davi, and Steffen Schulz. 2011. Scalable trust establishment with software reputation. In *Proceedings of the sixth ACM workshop on Scalable trusted computing*. ACM, 15–24.

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

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

[14] Nigel Edwards and Liqun Chen. 2012. An historical examination of open source releases and their vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 183–194.

[15] Jim Finkle and Supriya Kurane. 2014. U.S. hospital breach biggest yet to exploit Heartbleed bug: expert. *Reuters* (2014).

[16] Andy Greenberg. 2014. Hackers are already using the shellshock bug to launch botnet attacks. *Wired* (2014).

[17] HackerOne. 2017. The Hacker-powered security report 2017. https://www.hackerone.com/resources/hacker-powered-security-report

[18] Munawar Hafiz and Ming Fang. 2016. Game of detections: how are security vulnerabilities discovered in the wild? *Empirical Software Engineering* 21, 5 (2016), 1920–1959.

[19] Jinyoo Kim, Yashwant K Malaiya, and Indrakshi Ray. 2007. Vulnerability discovery in multi-version software systems. In *High Assurance Systems Engineering Symposium, 2007. HASE'07. 10th IEEE*. IEEE, 141–148.

[20] Meir M Lehman. 1980. Programs, life cycles, and laws of software evolution. *Proc. IEEE* 68, 9 (1980), 1060–1076.

[21] Meir M Lehman, Juan F Ramil, Paul D Wernick, Dewayne E Perry, and Wladyslaw M Turski. 1997. Metrics and laws of software evolution-the nineties view. In *Proceedings Fourth International Software Metrics Symposium*. IEEE, 20–32.

[22] Nancy G. Leveson. 2009. Software Challenges in Achieving Space Safety. *Journal of the British Interplanetary Society (JBIS)* (2009).

[23] Frank Li and Vern Paxson. 2017. A large-scale empirical study of security patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2201–2215.

[24] Thomas Maillart, Mingyi Zhao, Jens Grossklags, and John Chuang. 2017. Given enough eyeballs, all bugs are shallow? Revisiting Eric Raymond with bug bounty programs. *Journal of Cybersecurity* 3, 2 (2017), 81–90.

[25] Antonio Nappa, Richard Johnson, Leyla Bilge, Juan Caballero, and Tudor Dumitras. 2015. 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*. 692–708. https://doi.org/10.1109/SP.2015.48

[26] Matus Nemec, Dusan Klinec, Petr Svenda, Peter Sekan, and Vashek Matyas. 2017. Measuring popularity of cryptographic libraries in internet-wide scans. In *Proceedings of the 33rd Annual Computer Security Applications Conference*. ACM, 162–175.

[27] Andy Ozment and Stuart E Schechter. 2006. Milk or wine: does software security improve with age?. In *USENIX Security Symposium*. 93–104.

[28] James Andrew Ozment. 2007. *Vulnerability discovery & software security*. Ph.D. Dissertation. University of Cambridge.

[29] Jianfeng Pan, Guanglu Yan, and Xiaocao Fan. 2017. Digtool: A Virtualization-Based Framework for Detecting Kernel Vulnerabilities. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 149–165.

[30] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. 2015. 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*. ACM, 426–437.

[31] Eric Rescorla. 2005. Is finding security holes a good idea? *IEEE Security & Privacy* 3, 1 (2005), 14–19.

[32] Yaman Roumani, Joseph K Nwankpa, and Yazan F Roumani. 2015. Time series modeling of vulnerabilities. *Computers & Security* 51 (2015), 32–40.

[33] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. 2017. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 167–182.

[34] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A fast address sanity checker. In *Presented as part of the 2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*. 309–318.

[35] Richard Stallman et al. 1991. Gnu general public license. *Free Software Foundation, Inc., Tech. Rep* (1991).

[36] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *23nd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*.

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

[38] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. 2019. *ERIM*: Secure, Efficient In-process Isolation with Protection Keys (*MPK*). In *28th USENIX Security Symposium (USENIX Security 19)*. 1221–1238.

[39] Cynthia Wagner, Alexandre Dulaunoy, Gérard Wagener, and Andras Iklody. 2016. 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*. ACM, 49–56.

[40] Walter Willinger, Vern Paxson, and Murad S Taqqu. 1998. Self-similarity and heavy tails: Structural modeling of network traffic. *A practical guide to heavy tails: statistical techniques and applications* 23 (1998), 27–53.

[41] Chaowei Xiao, Armin Sarabi, Yang Liu, Bo Li, Mingyan Liu, and Tudor Dumitras. 2018. 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*. 903–918. https://www.usenix.org/conference/usenixsecurity18/presentation/xiao

[42] Fabian Yamaguchi, Markus Lottmann, and Konrad Rieck. 2012. Generalized vulnerability extrapolation using abstract syntax trees. In *Proceedings of the 28th Annual Computer Security Applications Conference*. ACM, 359–368.

[43] Mingyi Zhao, Jens Grossklags, and Peng Liu. 2015. An empirical study of web vulnerability discovery ecosystems. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1105–1117.

## 11 ADDITIONAL FIGURES

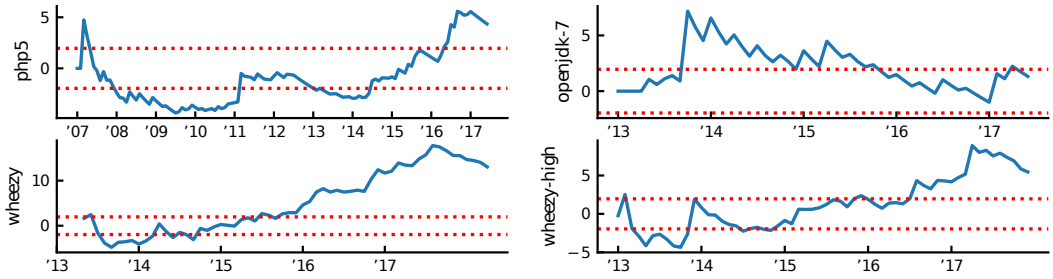



Fig. 15. Laplace trend tests with 95% significance thresholds (dashed lines).

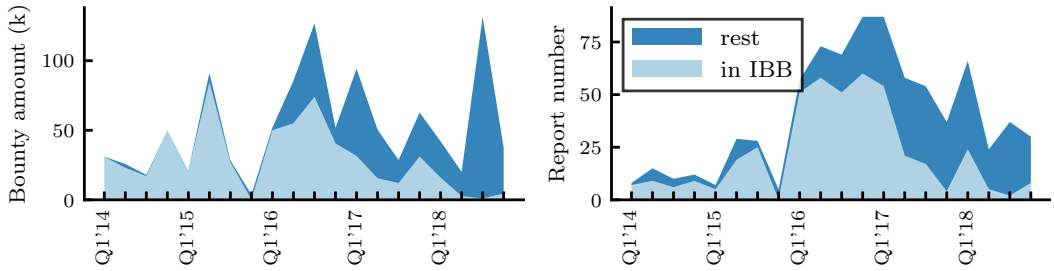



Fig. 16. Bounty amounts in thousands of USD (left) and number of reports (right) of IBB reporters (at least 1 IBB report at some point in time) comparing reports in the IBB program against reports for other programs in HackerOne over time.

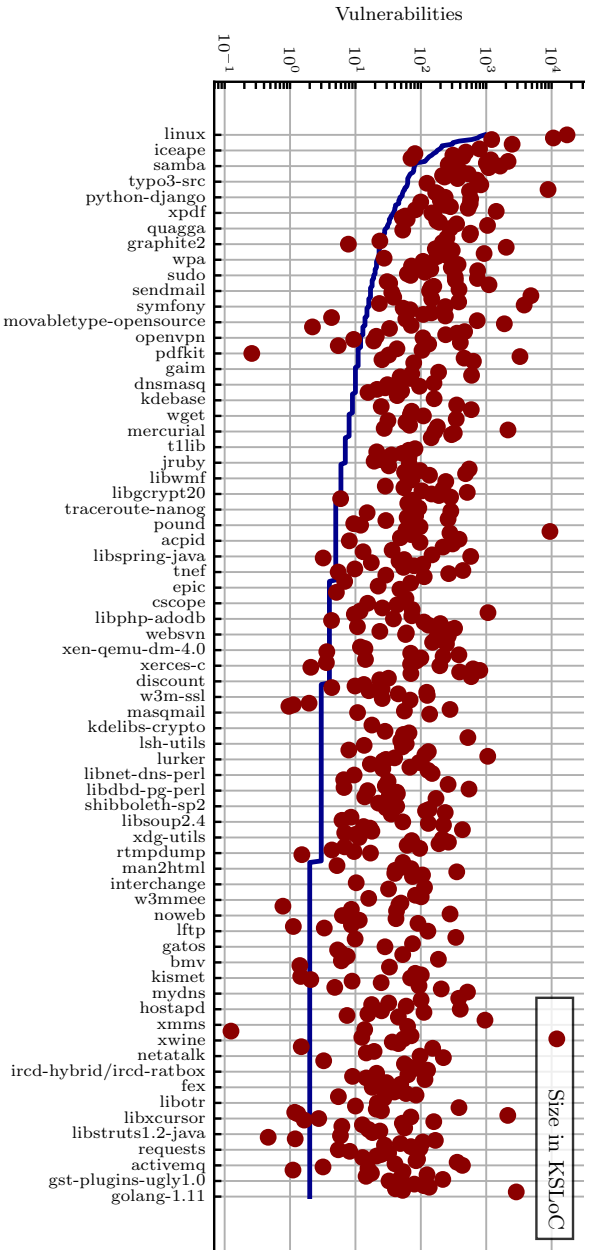



Fig. 17. The distribution of vulnerabilities in the Debian ecosystem (years 2001-2017), along with the size of the corresponding packages. The scale of axis x is logarithmic. All packages are taken into account. Every tenth package name appears on the y axis for space reasons.

## 12 STATISTICAL TEST RESULTS

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

Table 6. OLS for reliability trends

**Trend of the total number of vulnerabilities in Debian (Fig. 3b).**

| | | | |
|---|---|---|---|
| **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. 3c).**

| | | | |
|---|---|---|---|
| **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. 6).**

| | | | |
|---|---|---|---|
| **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. 9).**

| | | | |
|---|---|---|---|
| **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 7. OLS for bug bounty trends (Fig. 13)

**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 |