Underproduction: An Approach for Measuring Risk in Open Source Software

Kaylea Champion
University of Washington
Email: kaylea@uw.edu

Benjamin Mako Hill
University of Washington
Email: makohill@uw.edu

***Abstract*—The widespread adoption of Free/Libre and Open Source Software (FLOSS) means that the ongoing maintenance of many widely used software components relies on the collaborative effort of volunteers who set their own priorities and choose their own tasks. We argue that this has created a new form of risk that we call 'underproduction' which occurs when the supply of software engineering labor becomes out of alignment with the demand of people who rely on the software produced. We present a conceptual framework for identifying relative underproduction in software as well as a statistical method for applying our framework to a comprehensive dataset from the Debian GNU/Linux distribution that includes 21,902 source packages and the full history of 461,656 bugs. We draw on this application to present two experiments: (1) a demonstration of how our technique can be used to identify at-risk software packages in a large FLOSS repository and (2) a validation of these results using an alternate indicator of package risk. Our analysis demonstrates both the utility of our approach and reveals the existence of widespread underproduction in a range of widely-installed software components in Debian.**

***Index Terms*—open source, FLOSS, FOSS, OSS, mining software repositories, commons-based peer production, software quality, risk, quantitative methods**

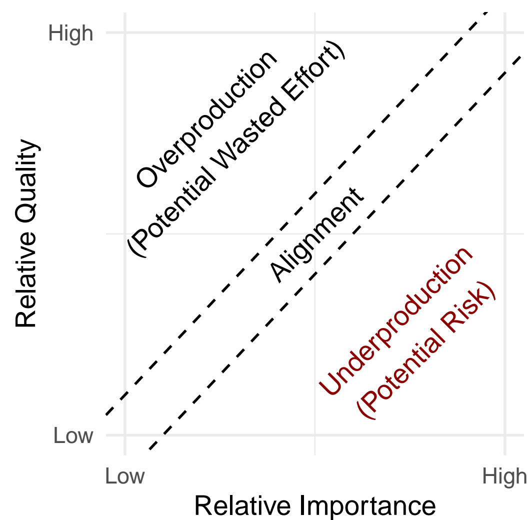

Fig. 1. A conceptual diagram locating underproduction in relation to quality and importance.

I. Introduction

In 2014, it was announced that the OpenSSL cryptography library contained a buffer over-read bug dubbed "Heartbleed" that compromised the security of a large portion of secure Internet traffic. The vulnerability resulted in the virtual theft of millions of health records, private government data, and more. OpenSSL provides the cryptographic code protecting a majority of HTTPS web connections, many VPNs, and variety of other Internet services. OpenSSL had been maintained through a "peer production" process common in Free/Libre and Open Source Software (FLOSS) where software development work is done by whomever is interested in taking on a particular task. For OpenSSL in early 2014, that had involved only four core developers, all volunteers. OpenSSL was at risk of an event like Heartbleed because it was an extraordinarily important piece of software with very little attention and labor devoted to its upkeep [1, 2]. In this paper, we describe an approach for identifying other important but poorly maintained FLOSS packages.

Over the last three decades, millions of people working in FLOSS communities have created an enormous body of software that has come to serve as digital infrastructure [3]. FLOSS communities have produced the GNU/Linux operating system, the Apache webserver, widely used development tools, and more [4]. In an early and influential practitioner account, Raymond argued that FLOSS would reach high quality through a process he dubbed "Linus' law" and defined as "given a large enough beta-tester and co-developer base, almost every problem will be characterized quickly and the fix obvious to someone" [5]. Benkler coined the term "peer production" to describe the method through which many small contributions from large groups of diversely motivated individuals could be integrated together into high quality information goods like software [6].

A growing body of research suggests reasons to be skeptical about Linus' law [7] and the idea that simply opening the door to one's code will attract a crowd of contributors [8, 9]. However, while a substantial portion of labor in many important FLOSS projects is paid [10], most important FLOSS projects are managed through peer production and continue to rely heavily on volunteer work [11]. Many FLOSS projects that incorporate paid labor have limited tools to coordinate or direct work, either paid or volunteer [11]. Although some FLOSS projects are now produced entirely within firms using traditional software development models, peer production

remains a critical feature of open source software production.

Over time, it has become clear that peer produced FLOSS projects' reliance on volunteer labor and self-selection into tasks has introduced types of risk that traditional software engineering processes have typically not faced. Foremost among these is what we call 'underproduction.' We use the term underproduction to refer to the fact that although a large portion of volunteer labor is dedicated to the most widely used open source projects, there are many places where the supply of quality software and volunteer labor is far out of alignment with demand. Because underproduction may go unnoticed or unaddressed until it is too late, we argue that it represents substantial risk to the stability and security of software infrastructure. As a result, we set our key research question as: *How can we measure underproduction in FLOSS?*, which we seek to answer both conceptually and empirically.

Our paper contributes to software engineering research in three distinct ways. First, we describe a broad conceptual framework to identify relative underproduction in peer produced FLOSS repositories: identifying software packages of lower relative quality than one would expect given their relative popularity. Second, we describe an application of this conceptual framework to a dataset of 21,902 source packages from the Debian GNU/Linux distribution using measures derived from multilevel Bayesian regression survival models. Finally, we present results from two experiments. The first experiment identifies a pool of relatively underproduced software in Debian. The second experiment seeks to validate our application of our framework for identifying underproduction by correlating underproduction with an alternate indicator of risk.

The rest of our paper is structured as follows. We describe prior work on underproduction in §II and present our conceptual framework in §III. We then describe Debian, our empirical setting, in §IV and our approach for applying our framework to Debian in §V. We present the results of our two experiments in §VI. Finally, we identify significant threats to the validity of our work in §VII and potential implications of the work in §VIII before concluding in §IX.

II. Background

A. Detecting and Measuring Software Risk

Prevention, detection, and mitigation of risk in software development, maintenance, and deployment are the subject of substantial research interest [e.g., 12, 13, 14]. Risk detection and management techniques examine overall system safety, develop predictive models to prioritize effort (such as reliability growth models), and seek development techniques to make a project less error-prone and more fault tolerant [15]. One line of work in software quality analysis and risk detection seeks to identify issues by locating "bad smells" and anti-patterns. This includes code smells [16, 17], as well as architectural smells (ill-considered fundamental design decisions that may trouble the project later) [18] and community smells (early warning signs of problems in a community). Of particular interest to both software engineering researchers and practitioners are smells that are empirically related to failures [19].

A range of other strategies have been employed to measure risk. Code-level metrics look at complexity, change-prone code, or defect-prone code. Other approaches consider the extent that a codebase takes adequate preventative measures against risk, such as thorough testing [20]. Finally, multi-factor approaches, as in decision-support analysis, take a risk management point of view and incorporate organizational factors, management practices, and areas of potential risk throughout a project's lifecycle [21].

B. Peer Production and FLOSS

Free/Libre and Open Source Software (FLOSS) is software released under a license that allows unrestricted use, modification, redistribution, and collaborative improvements [22]. FLOSS began with the free software movement in the 1980s and its efforts to build the GNU operating system as a replacement for commercial UNIX operating systems [23]. Over time, free software developers discovered that their free licenses and practices of working openly supported new forms of mass collaboration and bug fixes [4].

'Peer production' is a term coined by Yochai Benkler to describe the model of organizing production discovered by FLOSS communities in the early 1990s that involved the mass aggregation of many small contributions from diversely motivated individuals. Benkler [9] defines peer production for online groups in terms of four criteria: (1) decentralized goal setting and execution, (2) a diverse range of participant motives, including non-financial ones, (3) non-exclusive approaches to property (e.g. copyleft or permissive licensing), and (4) governance through participation, notions of meritocracy, and charisma, rather than through property or contract. In Benkler's foundational account [6], archetypes of peer production include FLOSS projects like the GNU operating system or the Linux kernel as well as efforts like Wikipedia.

C. Systematic comparison of FLOSS in software repositories

The process of building and maintaining software is often collaborative and social, including not only code but code comments, commit messages, pull requests, and code reviews, as well as bug reporting, issue discussing, and shared problem-solving [24]. Non-code trace data may include signals of technical debt [25], signs that a given code commit contains bugs [26], or serve as indicators of committed developers, a high-quality software project, or a healthy, sustainable project community [27, 28, 29]. Prior research has found that digital trace data capturing online community activity can provide significant insight into the study of software [27].

These collaborative and social systems offer a data source for understanding both developer team productivity, as in Choudhary's [30] study of "collaboration bursts" as well as for analyzing macro-level dynamics of software production. For example, Gonzalez-Barahona et al. [31] used the repository of *glibc* as a site to evaluate Lehman's "laws of software evolution" including the law of organizational stability which

states that work rates on a system are constant. The team found that the laws are frequently not upheld in FLOSS, especially when effort from outside a core team is considered. This work suggests that the effort available to maintain a piece of FLOSS software may increase as it grows in popularity.

Prior studies have suggested that bug resolution rate is closely associated of a range of important software engineering outcomes, including codebase growth, code quality, release rate, and developer productivity [32, 33, 34]. By contrast, lack of maintenance activity as reflected in a FLOSS project's bug tracking system can be considered a sign of failure [35].

III. Conceptual Framework: Underproduction

Repositories of peer produced FLOSS are susceptible to what we call underproduction—a concept and term that we borrow from several Wikipedia researchers who use the term to describe a dynamic that emerges when volunteers self-select into tasks. In particular, we are inspired by a study of Wikipedia by Warncke-Wang et al. [36] who build off previous work by Gorbatâi [37] to formalize what Warncke-Wang calls the "perfect alignment hypothesis" (PAH). The PAH proposes that the most heavily used peer produced information goods (for Warncke-Wang et al., articles in Wikipedia) will be the highest quality, that the least used will be the lowest quality, and so on. In other words, the PAH proposes that if we rank peer production products in terms of both quality and importance—for example, in the simple conceptual diagram shown in Figure 1—the two ranked lists will be perfectly correlated. In Gorbatâi's terminology, misalignment such that quality is high but demand is low results in 'overproduction.' Peer produced goods are 'underproduced' when demand is high but quality is low.

In an economic market, supply and demand are said to be aligned through a price mechanism. Alignment is reached because lower demand decreases prices, which disincentivizes production and returns a market to equilibrium. Because there is no price signal in Wikipedia to bring consumer demands and producer supply into equilibrium, it is unsurprising that Wikipedia deviates substantially from the predictions of the PAH [36]. Indeed, "perfect alignment" serves not as a serious prediction of the relationship between Wikipedia articles' quality to the interests of the general public but as a baseline from which to identify lacunae in need of attention [36]. Research on Wikipedia has sought to characterize the negative impacts on information consumers from divergence from the PAH baseline and to identify sociological processes through which underproduction might emerge [36, 37].

Despite the central role that FLOSS plays in peer production, we know of no efforts to conceptualize or measure underproduction in software. We find this surprising for two reasons. First, widespread underproduction seems likely in FLOSS given that FLOSS is characterized by self-selection of software developers into tasks, varying motives among contributors, and the frequent absence of market forces for allocating producers' labor [4, 38, 39]. Second, the consequences of underproduction are particularly stark in FLOSS where popular software acts as infrastructure [2, 11]. A low quality Wikipedia article on an extremely popular subject seems likely to pose much less risk to society than a bug like the Heartbleed vulnerability described earlier which could occur when FLOSS is underproduced. In this way, underproduction in software reflects an important, if underappreciated, type of risk in FLOSS.

To answer our research question (*How can we measure underproduction in FLOSS?*) in conceptual terms, our approach to detecting underproduction in software is composed of five steps as follows:

1) Assemble a collection of software artifacts that can be consistently measured as described below. These might be software packages, modules, source files, etc.
2) Identify one or more measures of quality that can be recorded consistently and independently (perhaps repeatedly) across each software artifact in the collection.
3) Similarly, identify a measure of importance that can be recorded consistently and independently across the collection.
4) Propose an *ex ante* theoretical baseline relationship between the two measures that reflects alignment. Although this might involve any number of assumptions about an ideal relationship, this might also be a non-parametric claim that the relative ranking of artifacts in terms of quality and importance will be perfectly correlated.
5) Measure deviation from this theoretical baseline across artifacts.

The measure of deviation resulting from this process serves as our measure of (mis-)alignment between quality and importance (i.e., over- or underproduction).

In a sense, our conceptual approach involves laying out software packages on dimensions similar to those shown in Figure 1, empirically identifying an ideal relationship that reflects general alignment (i.e., the zone in the lower left to upper right diagonal), and then measuring deviation from that ideal. This basic conceptual framework can incorporate any number of ways of measuring quality and importance—both areas of active work in software engineering research. Our approach can be carried out using a range of techniques for identifying alignment including entirely non-parametric rank-based approaches, machine learning-based ordinal categorization, or parametric regression-based techniques.

IV. Empirical Setting

The first step of applying our conceptual framework involves assembling a collection of software artifacts. We draw our collection from the Debian GNU/Linux distribution which acts as the empirical setting for all of our experiments. GNU/Linux distributions are collections of software that have been integrated, configured, tested, and packaged with an installer. The contributor community producing the distribution focuses primarily on the production of packages and package management tools for managing the installation and updating of software products produced by others. Distributions like Debian play an important role in the FLOSS ecosystem and

are the way that the vast majority of GNU/Linux users install operating system software as well as most applications and their dependencies. With a community in operation since 1993, Debian is widely used and is the basis for other widely-used distributions like Ubuntu. Debian had more than 1,400 different contributors in 2020[1] and contains more than 20,000 of the most important and widely used FLOSS packages.

Debian provides detailed and consistently measured longitudinal data on all its packages and maintainers in the form of released databases, datasets, and APIs [40, 41, 42]. A body of research in software engineering has used this open data from Debian to understand a range of software development practices. The Debian distribution has served as a basis for applying techniques to detect and mitigate defects [34, 43], predict bugs and vulnerabilities [44], detect the evolution of package incompatibilities [45], predict component reuse [46], demonstrate code clone detection techniques [47], develop generalizable QA techniques for complex projects [41], investigate package dependencies [48], and as an example of an information processing network [49].

V. Application of Framework

A. Step 1: Assemble a collection of artifacts

Our unit of analysis is the Debian *source package*. Source packages are the format that Debian's package maintainers modify and publish, but they are not used directly by end-users. Instead, source packages are built by computers in a Debian network of "build daemons" into one or more *binary packages* that may, or may not, include architecture-specific executables. These binary packages are then distributed to end-users and installed on their computers. Debian also provides tools to allow users to download and build their own binary packages from corresponding source packages. A single source package may produce many binary packages. For examples, although it is an outlier, the Linux kernel source package produces up to 1,246 binary packages from its single source package (most are architecture specific subcollections of kernel modules).

The one-to-many relationship between source and binary packages presented a challenge for our analysis. Although our chosen measure of quality (bug resolution, described in §V-B) uses information stored at the level of the source package [50], our chosen measure of importance (installations, described in §V-C), is aggregated at the binary level. To map source packages to binary packages, we used the Debian snapshot database's public APIs[2] to identify all binary packages produced by all versions of every source package.

B. Step 2: Identify a measure of quality

The second step of our framework involves identifying a measure of quality for each Debian source package. Quality in software is difficult to measure and common strategies for measuring quality include analyzing bug counts [e.g. 51] or assessing code internal design using a series of heuristics [e.g. 17]. However, software engineering researchers have noted that the quantity of bugs reported against a particular piece of FLOSS may be more related to the number of users of a package [50, 52], or the level of effort being expended on bug-finding [1] in ways that limit its ability to serve as a clear signal of software quality. In fact, Walden [1] found that OpenSSL had a lower bug count before Heartbleed than after. Walden [1] argued that measures of project activity and process improvements are a more useful sign of community recovery and software quality than bug count.

Additionally, techniques to assess codebases using code design heuristics are not oriented to the work of a distribution development community. The primary focus of work in a community like Debian is on configuring and testing packages for interoperability, rather than making changes to internal code design. In applying our method to Debian for this study, we follow a series of authors who have argued for a focus on a community's effectiveness at resolving the inevitable issues that arise instead of artifact-focused measures [4, 11, 53, 54, 55, 56]. Specifically, our measure of quality is the speed at which bugs are resolved. We treat the difference between opening and closing times as the *time to resolution*. Time to resolution has been cited as an important measure of FLOSS quality by a series of software engineering scholars [4, 11, 32, 33]. Although there may be many reasons for protracted resolution time in a distribution (e.g. maintainer skill, task complexity, report quality, lack of resources, bugs in the underlying software package causing packaging problems), the unresolved bug still represents an issue for the distribution's maintainer community and a problem for end users, whatever the reason.

Calculating this measure requires interpretation of bug reports and resolution data which Debian tracks using a database where each transaction takes the form of a specially formatted e-mail message. We extract comprehensive data on bugs from the Debian Bug Tracking System which is also referred to as "debbugs" or the "BTS." After obtaining an archival copy of all bugs from the BTS, we queried the Ultimate Debian Database [41] to map bugs to packages. We parsed all actions (i.e., e-mail messages) associated with each bug into columnar data for analysis. Using BTS data, we identified the date and time when each bug was opened and closed. We treated the marking of a bug as "closed," "forwarded" (i.e., not solved by the maintainer but rather referred to the "upstream" development team for the package itself and therefore no longer Debian's responsibility), or "merged" (i.e., designated as a repeat report of an issue already in the database) as closed.

Our approach differs from the approach taken by Zerouali et al. [57] who measure bugs as closed based on when they are last modified. We diverge from their method because final modification to a bug typically occurs when a bug is archived through an automated process that occurs about 30 days after closure. In our examinations of the database, we found that this process can be inconsistent and that unarchiving can occur as a function of administrative processing as well as due to true

[1]https://contributors.debian.org/ (Archived: https://web.archive.org/web/20201107231239/https://contributors.debian.org/)

[2]https://snapshot.debian.org/ (Archived: https://perma.cc/HQW7-R4Y2)

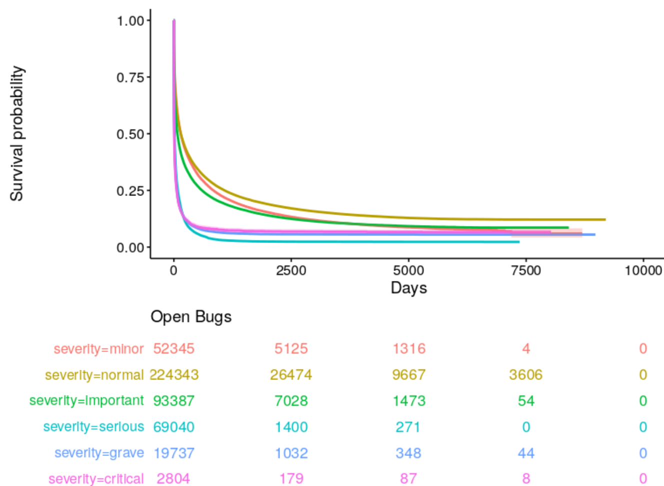

Fig. 2. A Kaplan-Meier curve that shows the number of bugs of different severities that remain open over time.

reopening of a bug.

Controlling for Bug Severity: A limitation relating to our measure of quality relates to the fact that not all bugs require equal attention. In recognition of this fact, bugs in Debian are assigned a *severity* by the submitter which can be modified by the package maintainer or others. Severity in Debian is one of the following categories: wishlist, minor, normal, important, serious, grave, and critical (in that order) with normal as the default. We extracted severity for every bug from the BTS to use as a control. We treat the 4,559 bugs where the severity is "not set" and the 2,239 bugs where the severity is listed as "fixed" as priority "normal."[3] We omit all "wishlist" severity bugs (124,961) because in addition to being used to track bugs that are "very difficult to fix due to major design considerations," this category may be used for a wide range of non-bug "to-do list" items.

Although there may be many other reasons for extended resolution time (e.g. skill, resources, inherent difficulty of the task), we do not include controls for these considerations. This is because our goal is to capture variation in resolution rate due to a range of reasons, which themselves may correspond to a level of risk. As a result, we at correspond to levels of risk. as a result we stratify on. describe threat and what we do to address the threat

Figure 2 shows non-parametric Kaplan-Meier survival curves for bugs of differing severities. These curves depict the probability of a given bug going from unresolved to resolved in days since it was filed. We observe in this plot that the curves are tightly clustered during the first few days after a bug is reported: about half of all bugs, regardless of severity, are solved shortly after they are reported. Bugs that are serious, grave, or critical are considered "release-critical" and must be fixed for the next "stable" release of Debian. Release-critical bugs include security problems, bugs that make packages unusable, or licensing requirements that are incompatible with Debian.[4] We observe that the curves in Figure 2 cluster into two general shapes that correspond to release critical and non-release critical bugs, and that release-critical bugs are fixed more quickly.

Modeling time to resolution using Bayesian survival models: Measuring quality as *time to resolution* poses two analytic challenges. The first challenge is the fact that a substantial portion of bugs remain unresolved at the point of data collection. Because bugs languishing in an open state is precisely the type of risk we hope to measure, omitting bugs that are open at the time of data collection would underestimate how quickly bugs are resolved. Following recent work in software engineering [32], we incorporate data on unresolved bugs by modeling the process of bug closure directly using Cox proportional hazard models.[5]

A second challenge in measuring quality as *time to resolution* comes from the fact that the distribution of bugs across packages is highly unequal. Most of the packages we examine (14,604 of 21,902) have 10 or fewer bugs and more than one out of six (3,857 of 21,902) have only one bug reported. In response, we use Bayesian hierarchical models fit using Stan. The intuition behind this choice is that when a package has few bugs, the overall problem resolution rate across Debian is more informative than the package's small number of data points. In general, the Bayesian approach allows us to become progressively more confident in an estimate when more bugs have been filed against a package.

Our approach finds support in the argument made by Ernst [58] who elaborates the value of Bayesian hierarchical modeling as a method for carefully distinguishing local and global characteristics in studies of software repositories. Our approach offers a "partial-pooling" model that allows us to both take advantage of the structure of our data (bugs are clustered within packages) and to update our prior assumption about expected resolution rates based on the new information that each bug supplies.

We incorporate our control for *severity* into a survival model of the following form where bugs are the unit of analysis and where λ reflects the hazard function capturing whether bug k in package j will be resolved at time t. We represent the severity for each bug as x_{jk} and use z_j to reflect the log of the package-level random effect for package j. [6]

$$\lambda(t; x_{jk}; q_j) = \lambda_0(t) exp(\beta x_{jk} + q_j)$$

Our measure of quality (q_j) is the package-level random effect of the posterior distributions in our survival models.

[3] The "fixed" severity was used from 2000 to 2002 to indicate a non-maintainer upload and then deprecated in favor of a "tagging" approach for this type of fix.

[4] https://www.debian.org/Bugs/Developer (Archived: https://perma.cc/MX3T-36PP)

[5] Cox models are a type of survival model developed originally in epidemiology to estimate how a behavior or treatment might prolong or shorten patients' lives while incorporating data from individuals for whom data is censored (i.e., individuals who are still alive at the conclusion of data collection). In our case, data is censored for any open bug.

[6] We use notation for random effects survival models drawn from Sargent [59].

We estimate this quantity using 4,000 independent draws from each package's posterior distribution using Stan. We take 95% credible intervals of these empirical distributions to reflect uncertainty. Estimates of q_j effects for all 21,902 packages are reported in our supplement described in §IX.

C. Step 3: Identify a measure of importance

Our third step involves identifying a consistent measure of importance for every artifact in our collection. In FLOSS contexts, importance can be measured as attention on hosting sites, number of active users, intensity of use, dependency networks, or criticality of function (e.g., life-safety systems) [11, 60, 61]. We measure importance using data from from the Debian "Popularity Contest" or "Popcon" application. Popcon is an opt-in survey that shares anonymous data from volunteer systems back with Debian. Popcon data has been used in a range of previous studies in software research [50, 52, 60]. Popcon is particularly applicable in our case because it is a signal which Debian itself has developed and deployed and because it is displayed in multiple locations in Debian's maintenance platforms. One important limitation of Popcon data is that despite the millions of systems running Debian, only a fraction have opted to report installation data.

Our study includes data from 201,484 systems from a single-day snapshot on July 6, 2020. Popcon includes two measures: *inst* for *installation* (i.e. the presence of a package on a machine), which serves as our measure of importance for subsequent analysis (i_j, for each package j); and a measure called "vote" which attempts to capture if packages are being used. In our analysis, we use *installation*; our supplement §IX includes a version of our analysis conducted using "vote" as our measure of importance.

Unfortunately, because Popcon reports installation at the binary level, we cannot use this data to distinguish whether a single individual reported the installation of multiple binaries associated with a given source package. To avoid double-counting, we use the binary-source mappings described in §V-A to set i_j to the largest install count among all binary packages associated with a source package j. As a result of this construction, our installation measure is necessarily conservative but we can be assured that a source package was installed at least i_j times.

D. Step 4: Select a baseline relationship

The fourth step in our conceptual framework involves comparing measures of quality and importance to some ideal baseline relationship. For our baseline, we take a non-parametric ranking approach that is similar to, but more granular than, the approach taken in Warncke-Wang et al.'s Wikipedia analysis which uses Wikipedia quality categories [36]. Our baseline definition of alignment is when the relative rankings of importance and quality are the same ($rq_i = rq_j$). We treat a ranking of 1 as describing the worst observed quality or lowest number of installations, while a rank of 21,902 represents the highest.

E. Step 5: Measure deviation

The final step of our conceptual framework involves measuring deviation from our theoretical baseline. We describe our measure of alignment as the "underproduction factor" (U_j) which we measure as the log of the ratio of rankings of importance and quality:

$$U_j = \log \frac{ri_j}{rq_j}$$

Given this construction, U_j will be zero when a package is fully aligned, negative if it is overproduced, and positive if it is underproduced. This approach means that the range of U_j is a function of repository size. In our dataset of 21,902 source packages, the range of $U_j = [-10, 10]$, where $U_j = 10$ for a theoretical maximally underproduced package where $ri_j = 21,902$ and $rq_j = 1$.

Although constructing U_j for a single value of q_j is straightforward, incorporating uncertainty in our measure of quality requires additional work. Because posterior draws in Stan are independent, we can incorporate uncertainty in our measure of quality by computing U_j using the quality ranking from each of 4,000 posterior draws taken from the estimated random effect for each package j. U_j reported in our analysis reflect the 95% credible intervals from U_j computed separately for these draws. Because we only have a single measure of installation i_j, we do not attempt to incorporate uncertainty in this measure into our analysis.

VI. Results from Experiments

In order to assess our conceptual model and to provide an empirical answer to our research question, we conduct two experiments. Our first experiment describes results from the application of our method described in §V and suggests that a minimum of 4,327 packages in Debian are underproduced. Our second experiment validates our approach using an alternate measure of risk.

A. Experiment 1: Identifying Underproduced Software

Our approach provides evidence that underproduction is widespread in Debian. Figure 3 shows 95% credible intervals (CIs) for all 21,902 packages. We describe packages whose 95% credible interval for U_j includes zero ($0 \in U_j$) as "aligned" and packages where both ends of the credible interval have the same sign as "overproduced" ($U_j < 0$) and "underproduced" ($U_j > 0$). The wide credible intervals for many of the packages shown on the left panel of Figure 3 reflect the fact that many of the aligned packages may be misaligned packages with high variance. The noise in our measures of q_j and U_j is partially attributable to the fact that many packages have few bugs.

Figure 4 displays a heatmap visualization that shows the number of packages occupying a range of installation ranks (ri_j) along the x-axis and quality ranks (rq_j) along the y-axis in evenly spaced bins. In this way, this non-parametric data visualization seeks to reflect the basic intuition that goes into the construction of our measure of underproduction U_j.

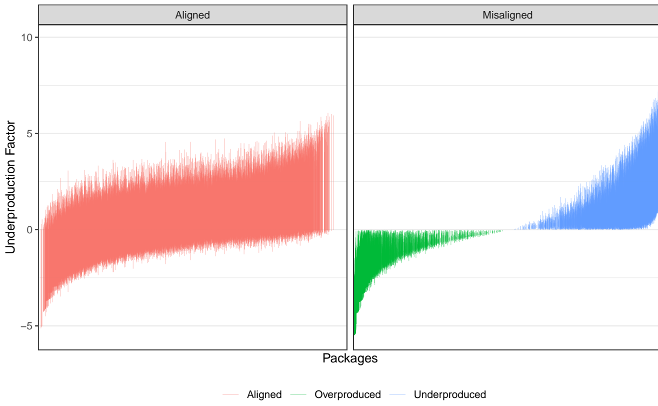

Fig. 3. Credible intervals for U_j for every package in Debian. We treat all packages whose CIs include zero as "aligned"; those whose CIs are entirely above 0 are labeled "underproduced;" those whose CIs are entirely below zero are labeled "overproduced."

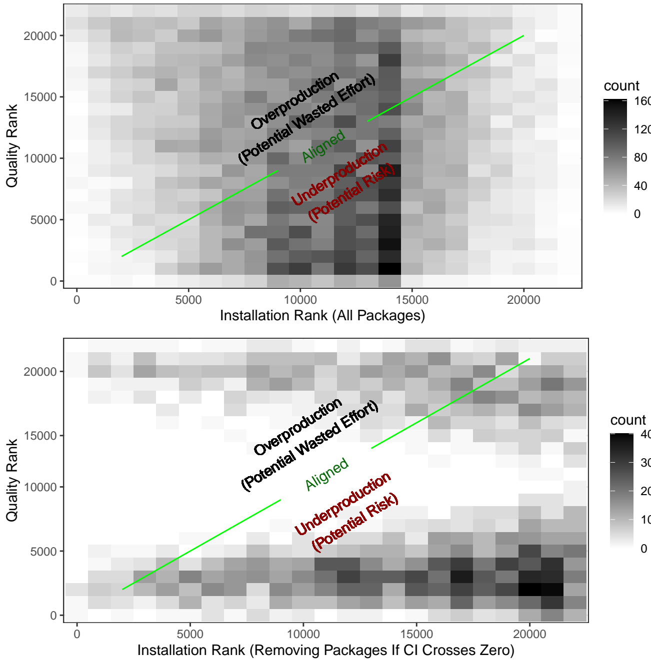

Fig. 4. A heatmap of software alignment. Color intensity indicates the number of packages occupying a given ranking of quality and installation. Aligned packages appear along the lower-left to upper-right diagonal. The top heatmap includes all packages, while the bottom heatmap contains only those packages for which the 95% credible interval does not cross zero.

Because rq_j is a distribution, the figures display a ranking of the mean value of q_j. The top panel shows all packages in Debian and clearly shows data spread across the heatmap rather than clustered along the diagonal. The relative lack of density along diagonal in the top figure is evidence that misalignment in Debian is widespread. The relatively high density of packages in the lower right indicates that underproduction in Debian is common.

The lower panel of Figure 4 shows only packages for which the 95% credible interval excludes zero (i.e., the misaligned packages shown on the right panel of Figure 3). Removing aligned packages reveals much more density among underproduced packages relative to overproduced packages. Removing 'aligned' packages visibly hollows out the center of the heatmap filled with packages of moderate quality because many of these packages have wide CIs. However, although there are almost no packages of middling quality that we can confidently say are overproduced, there are many that we can confidently say are underproduced. Further, there is substantial density—hundreds of packages—in the extreme bottom right corner. This area contains packages that are almost maximally underproduced according to our measure. In Figure 5, we present a list of the packages displaying the highest levels of underproduction (mean U_j) with per-package boxplots drawn from the posterior distribution associated with each package.

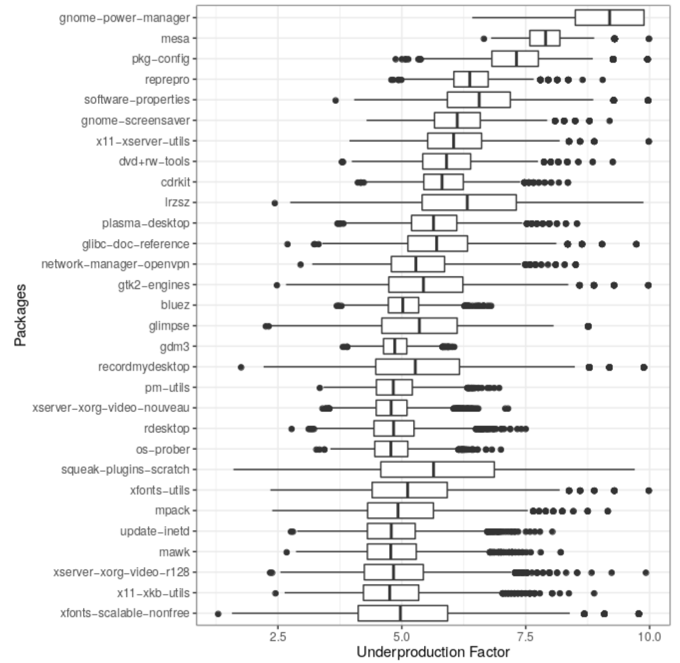

Fig. 5. Packages displaying the highest mean levels of underproduction. Boxplots show the mean and interquartile range of our distributions of U_j and reflect uncertainty in our model of package-level quality.

B. Experiment 2: Validation Using Alternate Indicator

To validate our application of our conceptual framework, we test whether our measure of underproduction can be used to predict community responses to risk. To do so, we collect data on the count of "non-maintainer uploads" (NMUs) which occur when any individual other than the denoted maintainer updates a version of a package in Debian.[7] Although some package maintainers may welcome NMUs, or even invite them, NMUs are widely understood as an indicator of risk in Debian. One of the criteria for "package salvage," that is, the taking over of maintainership without cooperation from

[7]https://wiki.debian.org/NonMaintainerUpload (Archived: https://perma.cc/TYN9-WQ79)

TABLE I
PREDICTING NON-MAINTAINER UPLOAD (NMU).

Intercept	-3.03^{*}
	$[-3.10; -2.95]$
Mean U_j	0.47^{*}
	$[0.40; 0.53]$
Num. obs.	21902

* 0 outside the confidence interval.

the designated maintainer, is the presence of NMUs.[8] We hypothesize that any valid measure of underproduction in Debian will be positively associated with the number of NMUs that the package receives.

Given that our measure of NMUs is an overdispersed count, we conduct our analysis using a negative binomial regression framework. Results from our model are reported in Table I. We find that U_j (underproduction factor) is a statistically significant predictor of NMU count, and that increases in underproduction factor correlate with increased numbers of NMUs. Our model estimates suggest that there is a strong relationship between our measure of underproduction and NMU count ($\beta = 0.47$; CI $= [0.40, 0.53]$). To illustrate this effect, consider two prototypical packages: a fully aligned package ($U_j = 0$) and one of the most underproduced packages in the distribution as depicted in Figure 5 ($U_j = 5$). Our model predicts that, on average, a prototypical aligned package will be NMUed extremely rarely ($\widehat{\mathrm{NMU}} = 0.048$; or less than 5% of packages will have a single NMU). On the other hand, it suggests that our prototypical underproduced package will be NMUed as often as not ($\widehat{\mathrm{NMU}} = 0.504$).

VII. THREATS TO VALIDITY

Our experimental findings may be limited in their generalizability. Although Debian is widely used in software engineering research and the study of distributions has been identified as preferable to code repository hosting platforms [62], it is only a single empirical setting. Additionally, Debian is idiosyncratic in ways that might threaten our ability to generalize. For example, Debian's policy to only include FLOSS means we do not assess widely-installed packages with nonfree licenses or any tools that are not packaged in Debian. Additionally, although our sample captures a broad timespan of bug reports in Debian, it only reflects one snapshot in time for package installation. As a result, packages may have changed in importance over time in ways that our analysis does not capture. We also cannot know how our experimental findings might generalize to smaller, newer, more commercial, or more narrowly focused development communities.

With respect to internal validity, it is important to note that our results are only correlational. While prior research has implicated lack of maintenance as increasing the likelihood of failure, we do not develop evidence to support a causal claim [e.g. 1, 2, 28]. We hope that further work will demonstrate whether underproduction is predictive of significant failure. The validity of our approach is also subject to threats related to construct validity. Underproduction is a concept borrowed from economics and involves a relationship between supply and demand. Although we have leveraged existing studies of underproduction in Wikipedia as part of our process of conceptualization, there remains space for further discussion about what should constitute 'supply' and 'demand' in FLOSS. For example, we treat supply as quality but it might also be conceptualized as code quantity or developer effort. Should demand be a raw measure of consumption, as we have conceptualized it, or should we explore alternate approaches like central positions in software dependency networks?

In a related sense, our empirical work is subject to threats that stem from choices we made in operationalization. For example, resolution time is an imperfect and partial measure of quality. Although we have omitted bugs that package maintainers reject, certain packages might receive higher numbers of low quality or difficult-to-reproduce bug reports from less technical users, prolonging resolution time. The very high bar to filing a bug in Debian may mitigate this concern.[9] Additionally, resolution time is limited in that it emphasizes the quality-in-use perspective rather than directly taking up artifact concerns (e.g., is the code written in ways that make it efficient to maintain). Applying artifact-based metrics from software engineering to GNU/Linux distributions like Debian remains an active area of research in software engineering. We welcome future attempts to integrate alternate metrics into our framework.

Our measure of installation is constrained to the systems whose administrators have volunteered data using Popcon. As with all opt-in surveys, this results in a non-random sample. Bias in this sample is possible because the installation of certain packages is possibly correlated with participation in the survey. It is also the case that importance and install base may be measured in different ways and our choice of metric may influence our results [63]. Although we consulted with Debian community members in designing and interpreting our experiments, there is no ground truth that we can use to ensure that we got it right.

Our application of our approach to underproduction analysis is limited in that the *ex ante* baseline we selected to demonstrate our approach relies on rank ordering and can only identify *relative* underproduction within a group of software components. If our method were applied to a collection of software components where all software was underproduced, it would be able to identify the worst of the batch but could not reveal a high degree of risk in general. Although this decision side-steps the need for parametric assumptions, an ordered ranking also means that we do not distinguish between consecutively ranked components.

Finally, we validated our approach using non-maintainer upload (NMU) count: the number of times a package is

[8] https://wiki.debian.org/PackageSalvaging (Archived: https://perma.cc/NBF9-6QSK)

[9] The onerous process of filing a bug report in Debian is described here: https://www.debian.org/Bugs/Reporting (Archived: https://perma.cc/RG3U-ZC7J).

updated by someone other than its maintainer. However, this measure is not an entirely independent measure: packages with long resolution times and high user bases may also be more likely to draw outside contributions in the form of NMUs.

VIII. Discussion

Our results suggest that underproduction is extremely widespread in Debian. Our non-parametric survival analysis shown in Figure 2 suggests that Debian resolves most bugs quickly and that release-critical bugs in Debian are fixed much more quickly than non-release-critical bugs. The presence of substantial underproduction in widely-installed components of Debian exposes Debian's users to risk. We explore several implications of these findings in the sections below.

A. The Long Tail of GUI Underproduction

One striking feature of our results is the predominance of visual and desktop-oriented components among the most underproduced packages (see Figure 5). Of the 30 most underproduced packages in Debian, 12 are directly part of the XWindows, GNOME, or KDE desktop windowing systems. For example, the "worst" ranking package, GNOME Power Manager (*gnome-power-manager*) tracks power usage statistics, allows configuration of power preferences, screenlocking, screensavers, and alerts users to power events such as an unplugged AC adaptor. Seven additional packages in Figure 5 are also oriented to desktop uses. For example, the *pm-utils* will suspend or hibernate a computer, *network-manager-openvpn* manages network connectivity through VPNs, Ethernet, and WiFi, *mesa* is a 3D graphics library, *bluez* is part of the Linux Bluetooth stack, *cdrkit* and *dvd+rw-tools* are both tools for creating CDs and DVDs, and *recordmydesktop* is a screen capture program. Our finding of relative underproduction in these programs appears to be in line with the history of critiques of GNU/Linux with respect to desktop usability and visual tools [64]. Although some of the reported issues in these GUI tools may be aesthetic, inspection reveals flaws reported at a range of severities including many very serious bugs.

A critique of the significance of this result might be that visual components are less likely to be used in business-critical circumstances. However, these packages have enormous install bases and are relied upon by many other packages. These results might simply reflect the difficulty of maintaining desktop-related packages. For example, maintaining *gnome-power-manager* includes complex integration work that spans from a wide range of low-level kernel features to high-level user-facing and usability issues. This pattern of underproduced GUI components suggests that although desktop GNU/Linux has made substantial progress, it remains a source of risk.

B. Implications for Software Engineering Research

Although our conceptual model and experiments demonstrate that underproduction can be measured, the detection and measurement of underproduction and the modeling of its predictors, causes, and remedies reflect a series of open challenges for the software engineering research community. More work is needed to further validate our conceptual framework and our statistical approach. We also hope that future work will extend our approach and implement our framework in other repositories of FLOSS.

C. Implications for Practice

FLOSS communities may find it useful to employ our technique to identify underproduced software in their repositories and to allocate resources and developer attention accordingly. For example, the Debian project has a volunteer QA team who might benefit from using our analysis to allocate its effort. Although underproduction is likely to be a particularly acute problem in FLOSS projects due to developer self-selection into tasks, it may also exist in non-FLOSS contexts. We look forward to working with managers of software repositories in both FLOSS and non-FLOSS contexts to help them implement and take action based on measures of underproduction.

While FLOSS practitioners may be particularly worried about underproduction in their projects, the software infrastructure risk that results from underproduction in FLOSS is of broader concern. FLOSS acts as global digital infrastructure. Failures in that infrastructure ripple through supply chains and across sectors. Technology leaders may see opportunities to improve their own risk profiles by offering support to FLOSS components identified as relatively underproduced through the type of analysis we have described.

Finally, many studies have examined the effect of funding when firms participate in FLOSS. Mixed and often ineffective results might be improved if underproduction is taken into account when directing resources to FLOSS projects. Tradeoffs associated with the potentially demotivating effect of money may be mitigated if funds are targeted at high-impact areas with little existing volunteer labor. The influx of successful investment that followed the Heartbleed vulnerability [1] may offer concerned organizations some hope that intervention in response to underproduction is possible and can be effective. Our work seeks to help guide these types of interventions before events like Heartbleed.

IX. Conclusion

Our work makes three important contributions to software engineering research: we present a broad conceptual framework for identifying relative underproduction; we illustrate our approach using data from Debian; and we validate our method using a measure of response to risk. Results from our experiments revealed significant underproduction in the widely used Debian distribution and suggest that many of the most underproduced packages in Debian are desktop applications.

Flaws in widely used software components, regardless of their purpose, represent a source of risk to our shared digital infrastructure. Even if a given bug does not result in system failure, it may provide an attack surface for intrusion or block upgrades of other vulnerable or failure-prone components. Despite widespread dependence on FLOSS, the burden of maintenance continues to fall on small teams of volunteers selecting their own tasks. Without fresh investment of skilled

and engaged participants, this public resource will remain at risk. As with Heartbleed, underproduction may not be recognized until it is too late. We hope that our work offers a step toward preventing these failures.

Online Supplement

Data, code, and supplemental information for this paper are available in the Harvard Dataverse at the following URL: https://doi.org/10.7910/DVN/PUCD2P

Acknowledgement

The authors gratefully acknowledge support from the Sloan Foundation through the Ford/Sloan Digital Infrastructure Initiative, Sloan Award 2018-11356. Wm Salt Hale of the Community Data Science Collective and Debian Developers Paul Wise and Don Armstrong provided valuable assistance in accessing and interpreting data. Rene Just at the University of Washington generously provided valuable insight and feedback. We are also grateful to our anonymous reviewers who pointed out opportunities for improvement. This work was conducted using the Hyak supercomputer at the University of Washington as well as research computing resources at Northwestern University.

References

[1] J. Walden, "The Impact of a Major Security Event on an Open Source Project: The Case of OpenSSL," in *17th International Conference on Mining Software Repositories*, May 2020.

[2] N. Eghbal, *Roads and Bridges: The Unseen Labor Behind Our Digital Infrastructure*. Ford Foundation, Jul. 2016.

[3] M. Asay, "The real number of open source developers," *InfoWorld*, Nov. 2019. [Online]. Available: https://www.infoworld.com/article/3452881/the-real-number-of-open-source-developers.html

[4] K. Crowston, K. Wei, J. Howison, and A. Wiggins, "Free/Libre open-source software development: What we know and what we do not know," *ACM Computing Surveys*, vol. 44, no. 2, pp. 1–35, Feb. 2012.

[5] E. S. Raymond, *The cathedral and the bazaar: Musings on Linux and open source by an accidental revolutionary*, T. O'Reilly, Ed. Sebastopol, CA: O'Reilly and Associates, 1999.

[6] Y. Benkler, "Coase's penguin, or, Linux and 'The nature of the firm'," *The Yale Law Journal*, vol. 112, no. 3, p. 369, Dec. 2002.

[7] C. M. Schweik, R. C. English, M. Kitsing, and S. Haire, "Brooks' Versus Linus' Law: An Empirical Test of Open Source Projects," in *Proceedings of the 2008 International Conference on Digital Government Research*. Montreal, Canada: Digital Government Society of North America, 2008, pp. 423–424.

[8] C. M. Schweik and R. C. English, *Internet success: a study of open-source software commons*. Cambridge, MA: MIT Press, 2012.

[9] Y. Benkler, A. Shaw, and B. M. Hill, "Peer production: a form of collective intelligence," in *Handbook of Collective Intelligence*, T. W. Malone and M. S. Bernstein, Eds. Cambridge, MA: MIT Press, 2015, pp. 175–204.

[10] M. Germonprez, J. Lipps, and S. Goggins, "The rising tide: Open source's steady transformation," *First Monday*, vol. 24, no. 8, Aug. 2019.

[11] N. Eghbal, *Working In Public: The Making and Maintenance of Open Source Software*. San Francisco, California: Stripe Press, 2020.

[12] R. Natella, D. Cotroneo, and H. S. Madeira, "Assessing Dependability with Software Fault Injection: A Survey," *ACM Computing Surveys*, vol. 48, no. 3, pp. 1–55, Feb. 2016.

[13] D. Gritzalis, G. Iseppi, A. Mylonas, and V. Stavrou, "Exiting the Risk Assessment Maze: A Meta-Survey," *ACM Computing Surveys*, vol. 51, no. 1, pp. 1–30, Apr. 2018.

[14] A. Meidan, J. A. García-García, I. Ramos, and M. J. Escalona, "Measuring Software Process: A Systematic Mapping Study," *ACM Computing Surveys*, vol. 51, no. 3, pp. 1–32, Jul. 2018.

[15] J. C. Bennett, G. A. Bohoris, E. M. Aspinwall, and R. C. Hall, "Risk analysis techniques and their application to software development," *European Journal of Operational Research*, vol. 95, no. 3, pp. 467–475, Dec. 1996.

[16] E. V. d. P. Sobrinho, A. De Lucia, and M. d. A. Maia, "A systematic literature review on bad smells 5 W's: which, when, what, who, where," *IEEE Transactions on Software Engineering*, 2018.

[17] J. A. M. Santos, J. B. Rocha-Junior, L. C. Lins Prates, R. S. do Nascimento, M. F. Freitas, and M. G. de Mendonca, "A systematic review on the code smell effect," *Journal of Systems and Software*, vol. 144, pp. 450–477, Oct. 2018.

[18] D. M. Le, D. Link, A. Shahbazian, and N. Medvidovic, "An Empirical Study of Architectural Decay in Open-Source Software," in *2018 IEEE International Conference on Software Architecture (ICSA)*, Apr. 2018, pp. 176–17 609.

[19] D. A. A. Tamburri, F. Palomba, and R. Kazman, "Exploring Community Smells in Open-Source: An Automated Approach," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.

[20] W. E. Wong, Y. Qi, and K. Cooper, "Source code-based software risk assessing," in *Proceedings of the 2005 ACM symposium on Applied computing - SAC '05*. Santa Fe, New Mexico: ACM Press, 2005, p. 1485.

[21] M. Pasha, G. Qaiser, and U. Pasha, "A Critical Analysis of Software Risk Management Techniques in Large Scale Systems," *IEEE Access*, vol. 6, pp. 12 412–12 424, 2018.

[22] K. Crowston, H. Annabi, J. Howison, and C. Masango, "Effective Work Practices for Software Engineering: Free/Libre Open Source Software Development," in *Proceedings of the 2004 ACM Workshop on Interdisciplinary Software Engineering Research*, New York, NY, USA, 2004, pp. 18–26.

[23] R. M. Stallman, *Free software, free society: Selected essays of Richard M. Stallman*, J. Gay, Ed., Oct. 2002. [Online]. Available: http://www.gnu.org/doc/Press-use/fsfs3-hardcover.pdf

[24] G. Robles, J. M. González-Barahona, D. Izquierdo-Cortazar, and I. Herraiz, "Tools for the Study of the Usual Data Sources found in Libre Software Projects:," *International Journal of Open Source Software and Processes*, vol. 1, no. 1, pp. 24–45, Jan. 2009.

[25] F. Zampetti, A. Serebrenik, and M. Di Penta, "Automatically Learning Patterns for Self-Admitted Technical Debt Removal," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. London, ON, Canada: IEEE, Feb. 2020, pp. 355–366.

[26] F. Falcao, C. Barbosa, B. Fonseca, A. Garcia, M. Ribeiro, and R. Gheyi, "On Relating Technical, Social Factors, and the Introduction of Bugs," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. London, ON, Canada: IEEE, Feb. 2020, pp. 378–388.

[27] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb, "Social coding in GitHub: transparency and collaboration in an open software repository," in *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work - CSCW '12*. Seattle, Washington, USA: ACM Press, 2012, p. 1277.

[28] J. Coelho, M. T. Valente, L. L. Silva, and E. Shihab, "Identifying unmaintained projects in github," in *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. Oulu Finland: ACM, Oct. 2018, pp. 1–10.

[29] M. Valiev, B. Vasilescu, and J. Herbsleb, "Ecosystem-level determinants of sustained activity in open-source projects: a case study of the PyPI ecosystem," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018*. Lake Buena Vista, FL, USA: ACM Press, 2018, pp. 644–655.

[30] S. Choudhary, C. Bogart, C. Rose, and J. Herbsleb, "Using Productive Collaboration Bursts to Analyze Open Source Collaboration Effectiveness," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. London, ON, Canada: IEEE, Feb. 2020, pp. 400–410.

[31] J. M. Gonzalez-Barahona, G. Robles, I. Herraiz, and F. Ortega, "Studying the laws of software evolution in a long-lived FLOSS project: Studying the Laws of Software Evolution," *Journal of Software: Evolution and Process*, vol. 26, no. 7, pp. 589–612, Jul. 2014.

[32] Z. Abou Khalil, E. Constantinou, T. Mens, L. Duchien, and C. Quinton, "A Longitudinal Analysis of Bug Handling Across Eclipse Releases," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Cleveland, OH, USA: IEEE, Sep. 2019, pp. 1–12.

[33] S. Kim and E. J. Whitehead, "How long did it take to fix bugs?" in *Proceedings of the 2006 international workshop on Mining software repositories - MSR '06*. Shanghai, China: ACM Press, 2006, p. 173.

[34] M. Michlmayr and A. Senyard, "A Statistical Analysis of Defects in Debian and Strategies for Improving Quality in Free Software Projects," in *The Economics of Open Source Software Development*. Elsevier, 2006, pp. 131–148.

[35] J. Coelho and M. T. Valente, "Why modern open source projects fail," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2017*. Paderborn, Germany: ACM Press, 2017, pp. 186–196.

[36] M. Warncke-Wang, V. Ranjan, L. Terveen, and B. Hecht, "Misalignment between supply and demand of quality content in peer production communities," in *Proceedings of the Ninth International AAAI Conference on Web and Social Media (ICWSM '15)*, 2015, pp. 493–502.

[37] A. D. Gorbatai, "Exploring Underproduction in Wikipedia," in *Proceedings of the 7th International Symposium on Wikis and Open Collaboration*, ser. WikiSym '11, 2011, pp. 205–206.

[38] K. R. Lakhani and B. Wolf, "Why hackers do what they do: Understanding motivation and effort in free/open source software projects," in *Perspectives on Free and Open Source Software*, J. Feller, B. Fitzgerald, S. A. Hissam, and K. R. Lakhani, Eds. MIT Press, 2005, pp. 3–22.

[39] M. O'Neil, *Cyberchiefs: Autonomy and Authority in Online Tribes*. Pluto Press, Nov. 2015.

[40] M. Caneill, D. M. German, and S. Zacchiroli, "The Debsources Dataset: Two Decades Of Free And Open Source Software," Aug. 2016. [Online]. Available: https://zenodo.org/record/61089

[41] L. Nussbaum and S. Zacchiroli, "The Ultimate Debian Database: Consolidating bazaar metadata for Quality Assurance and data mining," in *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. Cape Town, South Africa: IEEE, May 2010, pp. 52–61.

[42] A. Hindle, I. Herraiz, E. Shihab, and Zhen Ming Jiang, "Mining Challenge 2010: FreeBSD, GNOME Desktop and Debian/Ubuntu," in *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. Cape Town, South Africa: IEEE, May 2010, pp. 82–85.

[43] K. Chen and D. Wagner, "Large-scale analysis of format string vulnerabilities in Debian Linux," in *Proceedings of the 2007 workshop on Programming languages and analysis for security - PLAS '07*, 2007, p. 75.

[44] J. Pati and K. K. Shukla, "A comparison of ARIMA, neural network and a hybrid technique for Debian bug number prediction," in *2014 International Conference on Computer and Communication Technology (ICCCT)*. Allahabad, India: IEEE, Sep. 2014, pp. 47–53.

[45] M. Claes, T. Mens, R. Di Cosmo, and J. Vouillon, "A Historical Analysis of Debian Package Incompatibilities," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. Florence, Italy: IEEE, May 2015, pp. 212–223.

[46] S. Spaeth, G. von Krogh, M. Stuermer, and S. Haefliger, "A Lightweight Model of Component Reuse: A Study of Software Packages in Debian GNU/Linux," Feb. 2008. [Online]. Available: https://ssrn.com/abstract=1153968

[47] J. R. Cordy and C. K. Roy, "DebCheck: Efficient Checking for Open Source Code Clones in Software Systems," in *2011 IEEE 19th International Conference on Program Comprehension*. Kingston, ON, Canada: IEEE, Jun. 2011, pp. 217–218.

[48] J. A. Galindo Duarte, D. F. Benavides Cuevas, and S. Segura Rueda, "Debian Packages Repositories as Software Product Line Models. Towards Automated Analysis," in *First International Workshop on Automated Configuration and Tailoring of Applications*, 2010.

[49] P. Villegas, M. A. Munoz, and J. A. Bonachela, "Evolution in the Debian GNU/Linux software network: analogies and differences with gene regulatory networks," *Journal of The Royal Society Interface*, vol. 17, no. 163, p. 20190845, Feb. 2020.

[50] J. Davies, Hanyu Zhang, L. Nussbaum, and D. M. German, "Perspectives on bugs in the Debian bug tracking system," in *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. Cape Town, South Africa: IEEE, May 2010, pp. 86–89.

[51] B. Ray, D. Posnett, V. Filkov, and P. Devanbu, "A large scale study of programming languages and code quality in github," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*. Hong Kong, China: ACM Press, 2014, pp. 155–165.

[52] I. Herraiz, E. Shihab, T. H. Nguyen, and A. E. Hassan, "Impact of Installation Counts on Perceived Quality: A Case Study on Debian," in *2011 18th Working Conference on Reverse Engineering*. Limerick, Ireland: IEEE, Oct. 2011, pp. 219–228.

[53] E. Ronchieri and M. Canaparo, "Metrics for Software Reliability: a Systematic Mapping Study," *Journal of Integrated Design & Process Science*, vol. 22, no. 2, pp. 5–25, Apr. 2018.

[54] A. Adewumi, S. Misra, N. Omoregbe, B. Crawford, and R. Soto, "A systematic literature review of open source software quality assessment models," *SpringerPlus*, vol. 5, no. 1, pp. 1–13, Nov. 2016.

[55] C. Ruiz and W. N. Robinson, "Measuring Open Source Quality: A Literature Review," *International Journal of Open Source Software and Processes*, vol. 3, no. 3, pp. 48–65, Jul. 2011.

[56] A. Aksulu and M. Wade, "A Comprehensive Review and Synthesis of Open Source Research," *Journal of the Association for Information Systems*, vol. 11, no. 11/12, pp. 576–656, Nov. 2010.

[57] A. Zerouali, T. Mens, G. Robles, and J. M. Gonzalez-Barahona, "On the Relation between Outdated Docker Containers, Severity Vulnerabilities, and Bugs," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Hangzhou, China: IEEE, Feb. 2019, pp. 491–501.

[58] N. A. Ernst, "Bayesian hierarchical modelling for tailoring metric thresholds," in *Proceedings of the 15th International Conference on Mining Software Repositories - MSR '18*. Gothenburg, Sweden: ACM Press, 2018, pp. 587–591.

[59] D. J. Sargent, "A General Framework for Random Effects Survival Analysis in the Cox Proportional Hazards Setting," *Biometrics*, vol. 54, no. 4, p. 1486, Dec. 1998. [Online]. Available: https://www.jstor.org/stable/2533673?origin=crossref

[60] A. Wiggins, J. Howison, and K. Crowston, "Heartbeat: Measuring Active User Base and Potential User Interest in FLOSS Projects," in *Open Source Ecosystems: Diverse Communities Interacting*, C. Boldyreff, K. Crowston, B. Lundell, and A. I. Wasserman, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, vol. 299, pp. 94–104.

[61] G. Stergiopoulos, P. Kotzanikolaou, M. Theocharidou, G. Lykou, and D. Gritzalis, "Time-based critical infrastructure dependency analysis for large-scale and cross-sectoral failures," *International Journal of Critical Infrastructure Protection*, vol. 12, pp. 46–60, Mar. 2016.

[62] S. Spaeth, M. Stuermer, S. Haefliger, and G. von Krogh, "Sampling in Open Source Software Development: The Case for Using the Debian GNU/Linux Distribution," in *2007 40th Annual Hawaii International Conference on System Sciences (HICSS'07)*. Waikoloa, HI: IEEE, Jan. 2007.

[63] A. Zerouali, T. Mens, G. Robles, and J. M. Gonzalez-Barahona, "On the Diversity of Software Package Popularity Metrics: An Empirical Study of npm," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Hangzhou, China: IEEE, Feb. 2019, pp. 589–593.

[64] C. L. Paul, "A Survey of Usability Practices in Free/Libre/Open Source Software," in *Open Source Ecosystems: Diverse Communities Interacting*, C. Boldyreff, K. Crowston, B. Lundell, and A. I. Wasserman, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, vol. 299, pp. 264–273.