Detecting and Countering Software Supply Chain Risks — A Sociotechnical Perspective

Kaylea Champion – kaylea@uw.edu
https://www.kayleachampion.com
University of Washington

February 27, 2025

Heartbleed test

FAQ/status

There should not be false results anymore, only "Uh-Oh"s.
If there are problems, head to the FAQ

Enter the hostname of a server to test it for CVE-2014-0160.

yahoo.com Go!

yahoo.com IS VULNERABLE.

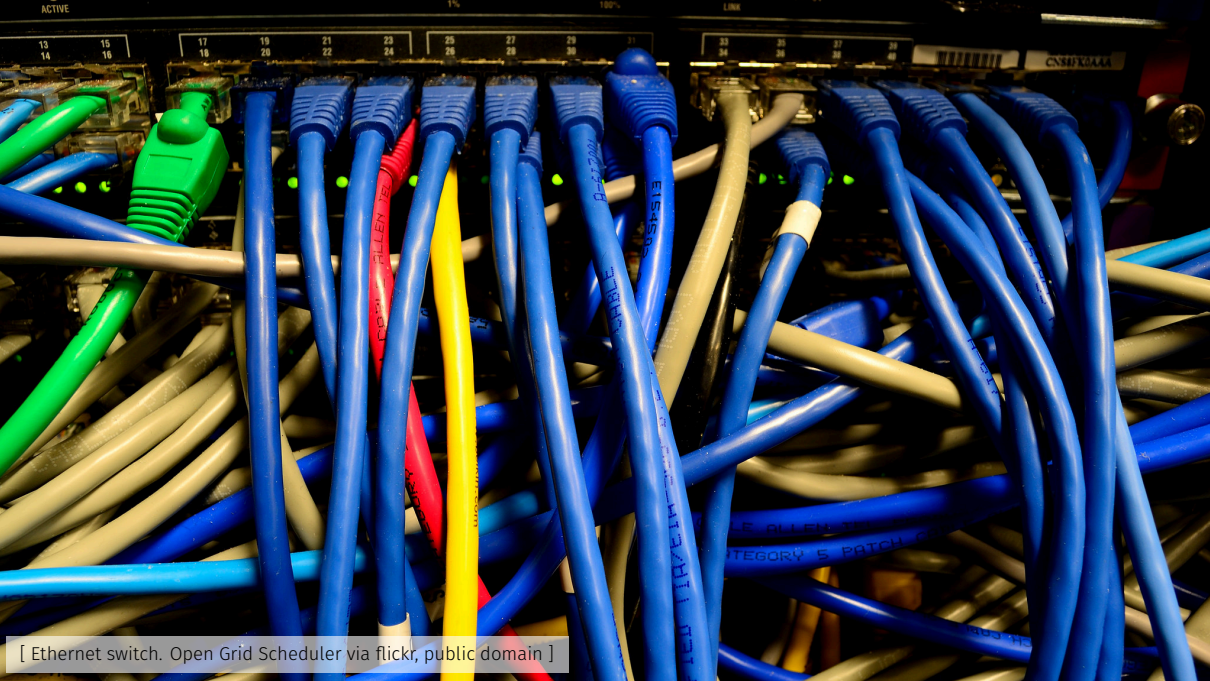
Here is some data we pulled from the server memory:

(we put `YELLOW SUBMARINE` there, and it should not have come back)

What is software supply chain risk?
How does it happen and what can we do about it?

[Showing the strain. Brian Smithson (@smithser) via flickr, CC BY 2.0]

[The 42 Domino Effect. Bernard Rose (br-images) via flickr, CC BY-NC 2.0]



[Ethernet switch. Open Grid Scheduler via flickr, public domain]

[Working. Cleber Quadros via flickr, CC BY-NC 2.0]

What is software supply chain risk?

What is software supply chain risk?

How does it happen and what can we do about it?

What is software supply chain risk?

Potential for failure due to misalignment of importance and quality/security.

How does it happen and what can we do about it?

What is software supply chain risk?

Potential for failure due to misalignment of importance and quality/security.

How does it happen and what can we do about it?

Due to technical and social factors which must be addressed simultaneously.

When I say “software supply chain,” I am pointing out that:

Global digital infrastructure software is developed and delivered through simultaneously decentralized and highly interdependent collaborations.

The “risk” I’m referring to is that:

This supply chain enables rapid innovation, but is also susceptible to disruption from multiple vectors: attack, errors, decay.

I propose that:

We can detect at-risk software
and intervene to prevent disruption.

Locating this line of work

- Increasing area of concern (academia, industry, government)

Locating this line of work

- Increasing area of concern (academia, industry, government)
- Problem space sits primarily in computing:

Locating this line of work

- Increasing area of concern (academia, industry, government)
- Problem space sits primarily in computing:
  - Engineering and development practices

Locating this line of work

- Increasing area of concern (academia, industry, government)
- Problem space sits primarily in computing:
  - Engineering and development practices
  - Maintenance processes

Locating this line of work

- Increasing area of concern (academia, industry, government)
- Problem space sits primarily in computing:
  - Engineering and development practices
  - Maintenance processes
  - Social computing / computer-supported cooperative work (CSCW)

Locating this line of work

- Increasing area of concern (academia, industry, government)
- Problem space sits primarily in computing:
  - Engineering and development practices
  - Maintenance processes
  - Social computing / computer-supported cooperative work (CSCW)
  - Cybersecurity practices

Locating this line of work

- Increasing area of concern (academia, industry, government)
- Problem space sits primarily in computing:
  - Engineering and development practices
  - Maintenance processes
  - Social computing / computer-supported cooperative work (CSCW)
  - Cybersecurity practices
  - Technical management structures

Locating this line of work

- Increasing area of concern (academia, industry, government)
- Problem space sits primarily in computing:
  - Engineering and development practices
  - Maintenance processes
  - Social computing / computer-supported cooperative work (CSCW)
  - Cybersecurity practices
  - Technical management structures
  - Technology policy and regulation

Locating this line of work

- Increasing area of concern (academia, industry, government)
- Problem space sits primarily in computing:
  - Engineering and development practices
  - Maintenance processes
  - Social computing / computer-supported cooperative work (CSCW)
  - Cybersecurity practices
  - Technical management structures
  - Technology policy and regulation
- A technology problem which requires sociotechnical analysis, explanation, and solutions.

Kaylea Champion

- Postdoc, Community Data Science Collective (UW, Northwestern, Purdue, UT Austin)

Kaylea Champion

- Postdoc, Community Data Science Collective (UW, Northwestern, Purdue, UT Austin)
- Research areas: computer-supported cooperative work, human-computer interaction, empirical software engineering, cybersecurity, privacy

Kaylea Champion

- Postdoc, Community Data Science Collective (UW, Northwestern, Purdue, UT Austin)
- Research areas: computer-supported cooperative work, human-computer interaction, empirical software engineering, cybersecurity, privacy
- 12+ publications (incl. ACM CSCW, IEEE SANER, IEEE Security & Privacy, ICSE CHASE) – I'll talk about 2 of these today in depth, with a 3rd in preparation

Kaylea Champion

- Postdoc, Community Data Science Collective (UW, Northwestern, Purdue, UT Austin)
- Research areas: computer-supported cooperative work, human-computer interaction, empirical software engineering, cybersecurity, privacy
- 12+ publications (incl. ACM CSCW, IEEE SANER, IEEE Security & Privacy, ICSE CHASE) – I'll talk about 2 of these today in depth, with a 3rd in preparation
- 25+ industry / community talks (practitioners, general public)

Kaylea Champion

- Postdoc, Community Data Science Collective (UW, Northwestern, Purdue, UT Austin)
- Research areas: computer-supported cooperative work, human-computer interaction, empirical software engineering, cybersecurity, privacy
- 12+ publications (incl. ACM CSCW, IEEE SANER, IEEE Security & Privacy, ICSE CHASE) – I'll talk about 2 of these today in depth, with a 3rd in preparation
- 25+ industry / community talks (practitioners, general public)
- PhD, Communication + Statistics Track, UW Seattle

Kaylea Champion

- Postdoc, Community Data Science Collective (UW, Northwestern, Purdue, UT Austin)
- Research areas: computer-supported cooperative work, human-computer interaction, empirical software engineering, cybersecurity, privacy
- 12+ publications (incl. ACM CSCW, IEEE SANER, IEEE Security & Privacy, ICSE CHASE) – I'll talk about 2 of these today in depth, with a 3rd in preparation
- 25+ industry / community talks (practitioners, general public)
- PhD, Communication + Statistics Track, UW Seattle
- MS, Computer Science, University of Chicago

Kaylea Champion

- Postdoc, Community Data Science Collective (UW, Northwestern, Purdue, UT Austin)
- Research areas: computer-supported cooperative work, human-computer interaction, empirical software engineering, cybersecurity, privacy
- 12+ publications (incl. ACM CSCW, IEEE SANER, IEEE Security & Privacy, ICSE CHASE) – I'll talk about 2 of these today in depth, with a 3rd in preparation
- 25+ industry / community talks (practitioners, general public)
- PhD, Communication + Statistics Track, UW Seattle
- MS, Computer Science, University of Chicago
- Industry experience: IT systems (career ladder: service delivery, system administration, management, leadership)

My approach

My research analyzes how software is built,

My approach

My research analyzes how software is built,

My approach

My research analyzes how software is built,

maintained,

My approach

My research analyzes how software is built,

maintained,

and decays –

My approach

My research analyzes how software is built,

maintained,

and decays –

including what practices lead to secure and sustainable projects,

My approach

My research analyzes how software is built,
maintained,
and decays –
including what practices lead to secure and sustainable projects,
who participates,

My approach

My research analyzes how software is built,
maintained,
and decays –
including what practices lead to secure and sustainable projects,
who participates,
and what can go wrong.

Overview

- Introductions — me, the problem space

Overview

- Introductions — me, the problem space
- Cross-sectional method for measuring underproduction (supply chain risk) [paper 1]

Overview

- Introductions — me, the problem space
- Cross-sectional method for measuring underproduction (supply chain risk) [paper 1]
- Sociotechnical factors associated with risk [paper 2]

Overview

- Introductions — me, the problem space
- Cross-sectional method for measuring underproduction (supply chain risk) [paper 1]
- Sociotechnical factors associated with risk [paper 2]
- Expanding into longitudinal methods [working paper]

Overview

- Introductions — me, the problem space
- Cross-sectional method for measuring underproduction (supply chain risk) [paper 1]
- Sociotechnical factors associated with risk [paper 2]
- Expanding into longitudinal methods [working paper]
- Interventions and areas of work + future

Much of our digital infrastructure is built through commons-based peer production:

a self-organizing mix of volunteer and paid contributors

Much of our digital infrastructure is built through commons-based peer production:

a self-organizing mix of volunteer and paid contributors

making self-selected modular contributions,

Much of our digital infrastructure is built through commons-based peer production:

a self-organizing mix of volunteer and paid contributors
making self-selected modular contributions,
freely available under an open license.


R
node
JS
Ubuntu
OpenSSL
Cryptography and SSL/TLS Toolkit
debian
NGINX
android

Heartbleed test

FAQ/status

There should not be false results anymore, only "Uh-Oh"s.
If there are problems, head to the FAQ

Enter the hostname of a server to test it for CVE-2014-0160.

yahoo.com Go!

yahoo.com IS VULNERABLE.

Here is some data we pulled from the server memory:

(we put `YELLOW SUBMARINE` there, and it should not have come back)

CYBERSECURITY ADVISORY

Mitigating Log4Shell and Other Log4j-Related Vulnerabilities

Last Revised: December 23, 2021

Alert Code: AA21-356A

Summary

XZ Utils backdoor

From Wikipedia, the free encyclopedia

In February 2024, a malicious backdoor was introduced to the Linux build of the xz utility within the liblzma library in versions 5.6.0 and 5.6.1 by an account using the name "Jia Tan".[b][4] The backdoor gives an attacker who possesses a specific Ed448 private key remote code execution capabilities on the affected Linux system. The issue has been given the Common Vulnerabilities and Exposures number CVE-2024-3094 and has been assigned a CVSS score of 10.0, the highest possible score.[5]

While xz is commonly present in most Linux distributions, at the time of discovery the backdoored version had not yet been widely deployed to production systems, but was present in development versions of major distributions.[6] The backdoor was discovered by the software developer Andres Freund, who announced his findings on 29 March 2024.[7]

XZ Utils backdoor

Previous XZ logo contributed by Jia

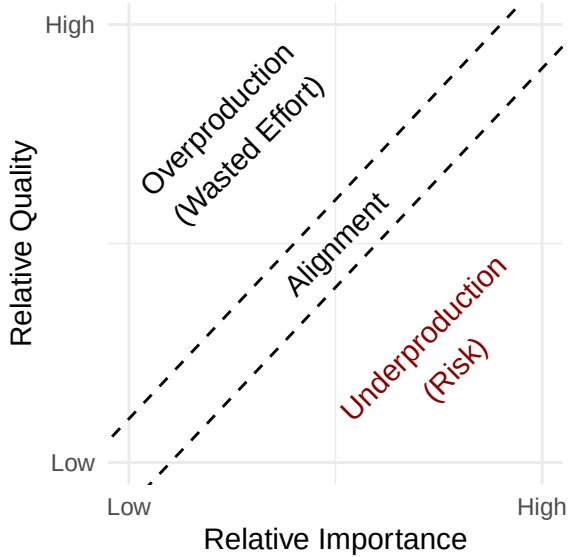
CVE identifier(s)	CVE-2024-3094
Date discovered	at or before 27 Ma 2024; 9 months ag [2]

[Literally crumbling infrastructure. J.C. Burns (jcburns) via flickr, CC BY-NC-ND 2.0]

Underproduction in digital infrastructure occurs when:
the quality of a good is relatively low
while its importance is relatively high.

[Champion and Hill (SANER 2021)]

Misalignment in FLOSS



[Champion and Hill (SANER 2021)]

Method: Underproduction Measurement

1. Identify a Body of Digital Infrastructure

[Champion and Hill (SANER 2021)]

Method: Underproduction Measurement

1. Identify a Body of Digital Infrastructure
2. Identify a Measure of Quality

[Champion and Hill (SANER 2021)]

Method: Underproduction Measurement

1. Identify a Body of Digital Infrastructure
2. Identify a Measure of Quality
3. Identify a Measure of Importance

[Champion and Hill (SANER 2021)]

Method: Underproduction Measurement

1. Identify a Body of Digital Infrastructure
2. Identify a Measure of Quality
3. Identify a Measure of Importance
4. Specify a Relationship Between Quality and Importance

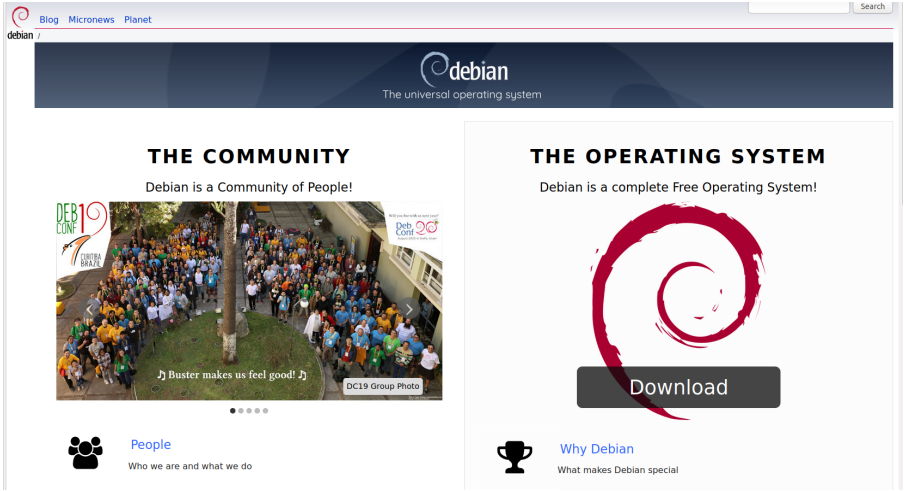
[Champion and Hill (SANER 2021)]

Method: Underproduction Measurement

1. Identify a Body of Digital Infrastructure
2. Identify a Measure of Quality
3. Identify a Measure of Importance
4. Specify a Relationship Between Quality and Importance
5. Test for Deviations to find Relative Underproduction

[Champion and Hill (SANER 2021)]

Step 1: Identify a Body of Digital Infrastructure



Sample size: 21,902 packages

[Champion and Hill (SANER 2021)]

Step 2: Identify a Measure of Quality

Bug resolution time as estimated through...

Bayesian Hierarchical Survival Analysis

[Champion and Hill (SANER 2021)]

Step 2: Identify a Measure of Quality

Bug resolution time as estimated through...

Bayesian Hierarchical Survival Analysis

[Champion and Hill (SANER 2021)]

Step 2: Identify a Measure of Quality

Bug resolution time as estimated through...

Bayesian Hierarchical Survival Analysis

[Champion and Hill (SANER 2021)]

Step 2: Identify a Measure of Quality

Bug resolution time as estimated through...

Bayesian Hierarchical Survival Analysis

[Champion and Hill (SANER 2021)]

Step 2 cont'd: Identify a Measure of Quality

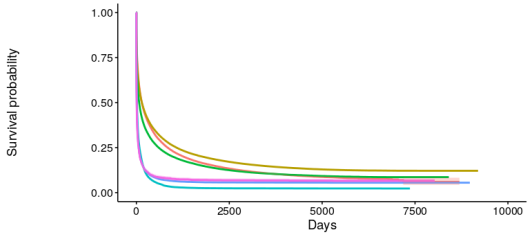
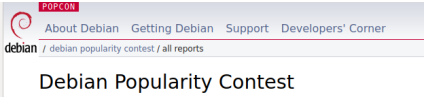


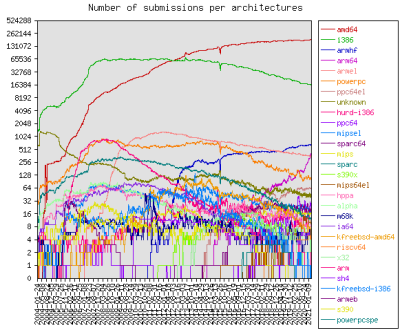
Figure 1: A Kaplan-Meier curve showing bugs of different severities that remain open over time.

Sample size: 461,656 bugs

[Champion and Hill (SANER 2021)]

Step 3: Identify a Measure of Importance

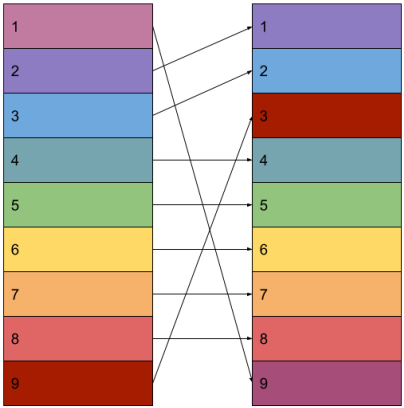




[Champion and Hill (SANER 2021)]

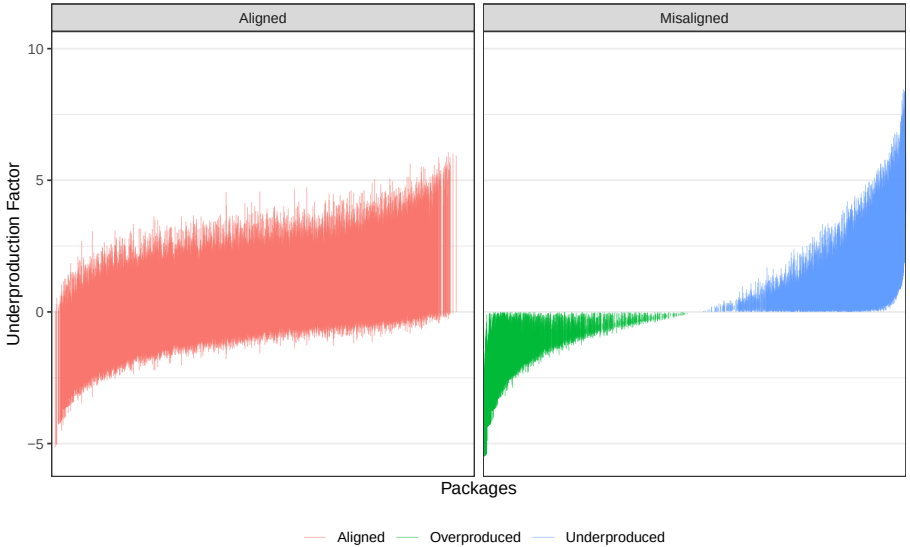
Sample size: 201,484 systems.

Step 4: Relating Quality and Importance



[Champion and Hill (SANER 2021)]

Step 5: Test for Deviations to Find Relative Underproduction



[Champion and Hill (SANER 2021)]

Validation: Non-maintainer Uploads

- A non-maintainer upload (NMU) is an indicator of risk
- Increased risk as measured through underproduction is associated with increased odds of a package receiving an NMU.

Table 1: Log-odds of receiving an NMU; U_j is underproduction factor.

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

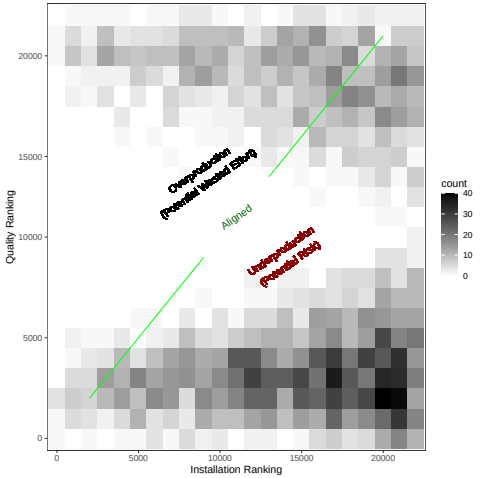
* 0 outside the confidence interval.

[Champion and Hill (SANER 2021)]

Contribution: New Method

1. Identify a Body of Digital Infrastructure
2. Identify a Measure of Quality
3. Identify a Measure of Importance
4. Specify a Relationship Between Quality and Importance
5. Test for Deviations to find Relative Underproduction

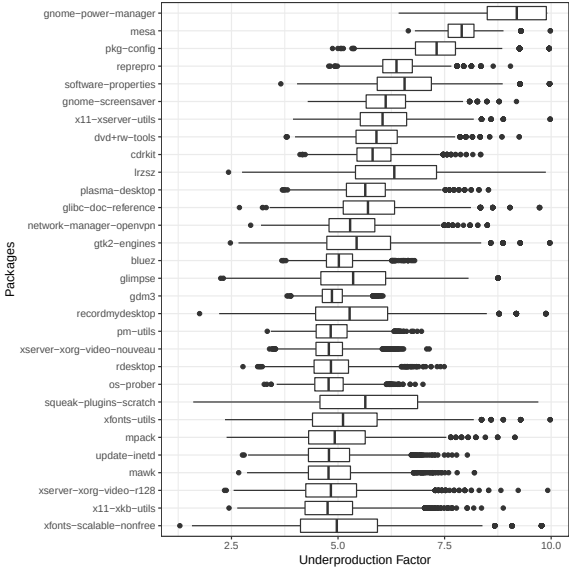
Contribution: Risk Quantification



Many widely used packages are low quality – the conditions for a problem like Heartbleed.

[Champion and Hill (SANER 2021)]

Contribution: Risk Quantification



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**

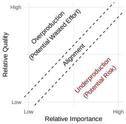


Fig. 1. A conceptual diagram locating underproduction in relation to quality and importance.

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

1

Impact:

- Sovereign Tech Fund, now Agency (German Gov't)

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 general purpose package ecosystem like 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 those 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, free/libre, peer production, software quality, risk, quantitative methods**

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

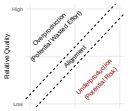


Fig. 1. A conceptual diagram locating underproduction in relation to quality and importance.

Impact:

- Sovereign Tech Fund, now Agency (German Gov't)
- Open SSF Criticality Score (Linux Foundation)

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

Impact:

- Sovereign Tech Fund, now Agency (German Gov't)
- Open SSF Criticality Score (Linux Foundation)
- A new angle for garnering insights

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

Impact:

- Sovereign Tech Fund, now Agency (German Gov't)
- Open SSF Criticality Score (Linux Foundation)
- A new angle for garnering insights
- Invited talks to practitioners, industry groups

When highly important software is low quality or not secure, we're all at risk. How does this happen?


Infrastructure
PODCAST
M
21
DEBCONF
ONLINE
DCG 201
CHAOSS
OpenSSF
SeaGL

How do important packages end up neglected?

Common suggestions....

- Age: Old packages, languages

[Champion and Hill (SANER 2024)]

How do important packages end up neglected?

Common suggestions....

- Age: Old packages, languages
- Lack of redundancy: too much reliance on single points of failure

[Champion and Hill (SANER 2024)]

How do important packages end up neglected?

Common suggestions....

- Age: Old packages, languages
- Lack of redundancy: too much reliance on single points of failure
- Isolated from the main community

[Champion and Hill (SANER 2024)]

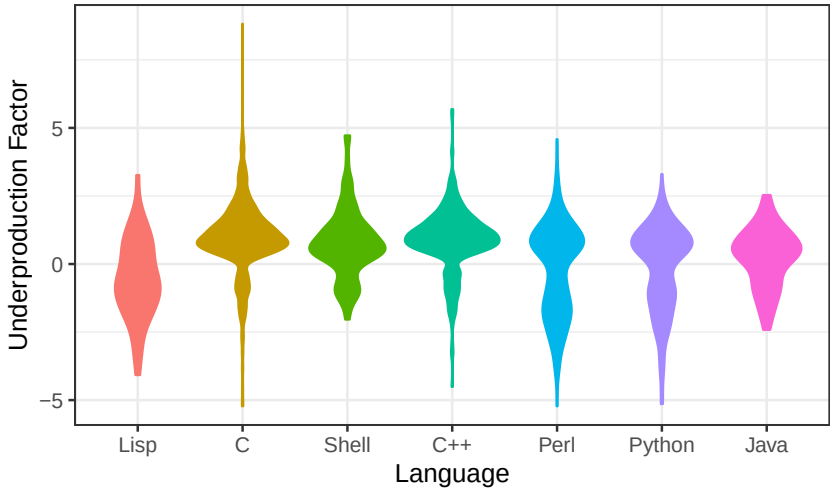
Hypotheses

- H1: Older packages are **more** likely to be underproduced.
- H2: Packages written in older languages are **more** likely to be underproduced.
- H3: Older packages in older languages (H1 + H2) are **more** likely to be underproduced.
- H4: Packages with more contributors are **less** likely to be underproduced.
- H5: Packages with less maintainer turnover are **less** likely to be underproduced.
- H6: Packages maintained by a team are **less** likely to be underproduced.
- H7: Packages with low influence (eigenvector centrality) are **more** likely to be underproduced.
- H8: Packages with low brokerage (betweenness centrality) are **more** likely to be underproduced.

H correct | H incorrect (sign flip) | Inconclusive / Not significant in full model

[Champion and Hill (SANER 2024)]

Results



[Champion and Hill (SANER 2024)]

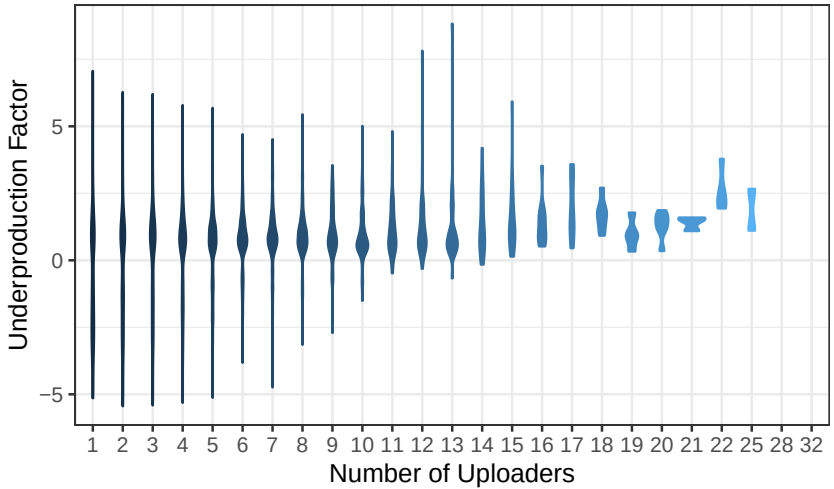
Hypotheses

- H1: Older packages are **more** likely to be underproduced.
- H2: Packages written in older languages are **more** likely to be underproduced.
- H3: Older packages in older languages (H1 + H2) are **more** likely to be underproduced.
- H4: Packages with more contributors are **less** likely to be underproduced.
- H5: Packages with less maintainer turnover are **less** likely to be underproduced.
- H6: Packages maintained by a team are **less** likely to be underproduced.
- H7: Packages with low influence (eigenvector centrality) are **more** likely to be underproduced.
- H8: Packages with low brokerage (betweenness centrality) are **more** likely to be underproduced.

H correct | H incorrect (sign flip) | Inconclusive / Not significant in full model

[Champion and Hill (SANER 2024)]

Uploaders



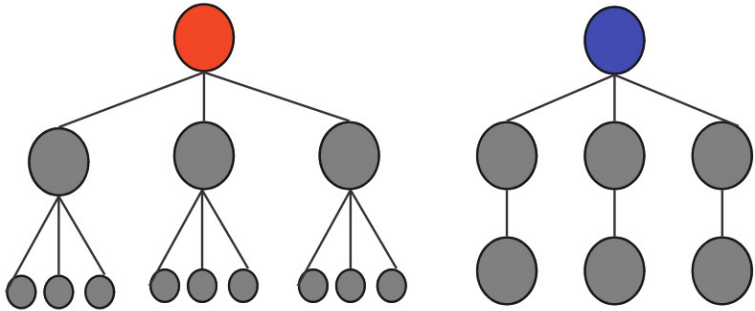
[Champion and Hill (SANER 2024)]

Hypotheses

- H1: Older packages are **more** likely to be underproduced.
- H2: Packages written in older languages are **more** likely to be underproduced.
- H3: Older packages in older languages (H1 + H2) are **more** likely to be underproduced.
- H4: Packages with more contributors are **less** likely to be underproduced.
- H5: Packages with less maintainer turnover are **less** likely to be underproduced.
- H6: Packages maintained by a team are **less** likely to be underproduced.
- H7: Packages with low influence (eigenvector centrality) are **more** likely to be underproduced.
- H8: Packages with low brokerage (betweenness centrality) are **more** likely to be underproduced.

H correct | H incorrect (sign flip) | Inconclusive / Not significant in full model

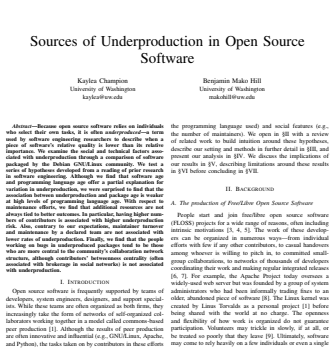
Influence



Packages with low **influence** (eigenvector centrality) are *less* likely to be underproduced (not more, as proposed!).

[Champion and Hill (SANER 2024)]

Takeaways



Sources of Underproduction in Open Source Software

Kaylea Champion
University of Washington
kaylea@uw.edu

Benjamin Mako Hill
University of Washington
makohill@uw.edu

Abstract—Because open source software relies on individuals who select their own tasks, it is often *underproduced*—a term used by software engineering researchers to describe when a piece of software's relative quality is lower than its relative importance. We examine the social and technical factors associated with underproduction in a population of open source packages packaged by the Debian GNU/Linux community. We test a series of hypotheses developed from a reading of prior research in software engineering. Although we find that software age and programming language age offer a partial explanation for variation in underproduction, we were surprised to find that the association between underproduction and package age is weaker at high levels of programming language age. With respect to maintenance efforts, we find that additional resources are not always helpful in preventing underproduction. Surprisingly, number of contributors is associated with higher underproduction risk. Also, contrary to our expectations, maintainer turnover and maintenance by a declared team are not associated with lower rates of underproduction. Finally, we find that the people working on bugs in underproduced packages tend to be those who are more central to the community's collaboration network structure, although contributors' betweenness centrality (often associated with brokerage in social networks) is not associated with underproduction.

I. Introduction

Open source software is frequently supported by teams of developers, system engineers, designers, and support specialists. While these teams are often organized as both firms, they increasingly take the form of networks of self-organized collaborators working together in a model called commons-based peer production [1]. Although the results of peer production are often innovative and influential (e.g., GNU/Linux, Apache, and Python), the tasks taken on by contributors in these efforts do not always align with tasks that are most needed by the software project's users or by the general public.

Of particular concern is software that is *underproduced*—i.e., low quality, but highly important. Underproduction has been shown to be widespread in open source software [2]. Some underproduced software may be buried deep in the software supply chain, and vulnerabilities and flaws may not be noticed until they cause disruptions. How can we identify underproduced software and remediate risks before they cause major failures?

This paper builds on previous software engineering research focused on risk measurement to test a series of hypotheses on correlates and theorized causes of underproduction in open source software oriented to both material conditions (e.g., the programming language used) and social features (e.g., the number of maintainers). We open in §II with a review of related work to build intuition around these hypotheses, describe our setting and methods in further detail in §III, and present our analysis in §IV. We discuss the implications of our results in §V, describing limitations around these results in §VI before concluding in §VII.

II. Background

A. The production of Free/Libre Open Source Software

People start and join free/libre open source software (FLOSS) projects for a wide range of reasons, often including intrinsic motivations [3, 4, 5]. The work of these developers can be organized in numerous ways—from individual efforts with few if any other contributors, to casual handovers among whoever is willing to pitch in, to committed small-group collaborations, to networks of thousands of developers coordinating their work and making regular integrated releases [6, 7]. For example, the Apache Project today oversees a widely-used web server but was founded by a group of system administrators who had been informally trading fixes to an older, abandoned piece of software [8]. The Linux kernel was created by Linus Torvalds as a personal project [1] before being shared with the world at no charge. The openness and flexibility of how work is organized do not guarantee participation. Volunteers may trickle in slowly, if at all, or be treated so poorly that they leave [9]. Ultimately, software may come to rely heavily on a few individuals or even a single person. Leaders can burn out and may not have a pipeline of candidates to assume key roles [10, 11].

Although FLOSS's impact can be large, the process that creates these goods can be inefficient. Benkler observed that some of the largest and most well-known FLOSS projects function as commons-based peer production organizations that rely on voluntary and self-organized labor [12]. Although a substantial portion of FLOSS is supported by firms [13], these firms do not generally assign tasks within FLOSS projects. Individuals—especially volunteers—tend to choose their own tasks. Unfortunately, the tasks individuals choose may not be those most needed by users. As a result, underproduction—when the quality of a good falls below its importance—can introduce an important form of risk.

- Age is important (but attenuates)

Takeaways

Sources of Underproduction in Open Source Software

Kaylea Champion
University of Washington
kaylea@uw.edu

Benjamin Mako Hill
University of Washington
makohill@uw.edu

***Abstract*—Because open source software relies on individuals who select their own tasks, it is often *underproduced*—a term used by software engineering researchers to describe when a piece of software's relative quality is lower than its relative importance. We examine the social and technical factors associated with underproduction through a comparison of software packaged by the Debian GNU/Linux community. We test a series of hypotheses developed from a reading of prior research in software engineering. Although we find that software age and programming language age offer a partial explanation for variation in underproduction, we were surprised to find that the association between underproduction and package age is weaker at high levels of programming language age. With respect to maintenance efforts, we find that additional resources are not always tied to better outcomes. In particular, having higher numbers of contributors is associated with higher underproduction risk. Also, contrary to our expectations, maintainer turnover and maintenance by a declared team are not associated with lower rates of underproduction. Finally, we find that the people working on bugs in underproduced packages tend to be those who are more central to the community's collaboration network structure, although contributors' betweenness centrality (often associated with brokerage in social networks) is not associated with underproduction.**

I. Introduction

Open source software is frequently supported by teams of developers, system engineers, designers, and support specialists. While these teams are often organized as both firms, they increasingly take the form of networks of self-organized collaborators working together in a model called commons-based peer production [1]. Although the results of peer production are often innovative and influential (e.g., GNU/Linux, Apache, and Python), the tasks taken on by contributors in these efforts do not always align with tasks that are most needed by the software project's users or by the general public.

Of particular concern is software that is *underproduced*—i.e., low quality, but highly important. Underproduction has been shown to be widespread in open source software [2]. Some underproduced software may be buried deep in the software supply chain, and vulnerabilities and flaws may not be noticed until they cause disruptions. How can we identify underproduced software and remediate risks before they cause major failures?

This paper builds on previous software engineering research focused on risk measurement to test a series of hypotheses on correlates and theorized causes of underproduction in open source software oriented to both material conditions (e.g., the programming language used) and social features (e.g., the number of maintainers). We open in §II with a review of related work to build intuition around these hypotheses, describe our setting and methods in further detail in §III, and present our analysis in §IV. We discuss the implications of our results in §V, describing limitations around these results in §VI before concluding in §VII.

II. Background

A. The production of Free/Libre Open Source Software

People start and join free/libre open source software (FLOSS) projects for a wide range of reasons, often including intrinsic motivations [3, 4, 5]. The work of these developers can be organized in numerous ways—from individual efforts with few if any other contributors, to casual handovers among whoever is willing to pitch in, to committed small-group collaborations, to networks of thousands of developers coordinating their work and making regular integrated releases [6, 7]. For example, the Apache Project today oversees a widely-used web server but was founded by a group of system administrators who had been informally trading fixes to an older, abandoned piece of software [8]. The Linux kernel was created by Linus Torvalds as a personal project [1] before being shared with the world at no charge. The openness and flexibility of how work is organized do not guarantee participation. Volunteers may trickle in slowly, if at all, or be treated so poorly that they leave [9]. Ultimately, software may come to rely heavily on a few individuals or even a single person. Leaders can burn out and may not have a pipeline of candidates to assume key roles [10, 11].

Although FLOSS's impact can be large, the process that creates these goods can be inefficient. Benkler observed that some of the largest and most well-known FLOSS projects function as commons-based peer production organizations that rely on voluntary and self-organized labor [12]. Although a substantial portion of FLOSS is supported by firms [13], these firms do not generally assign tasks within FLOSS projects. Individuals—especially volunteers—tend to choose their own tasks. Unfortunately, the tasks individuals choose may not be those most needed by users. As a result, underproduction—when the quality of a good falls below its importance—can introduce an important form of risk.

- Age is important (but attenuates)
- Individuals and small teams perform well...

Takeaways

Sources of Underproduction in Open Source Software

Kaylea Champion
University of Washington
kaylea@uw.edu

Benjamin Mako Hill
University of Washington
makohill@uw.edu

Abstract—Because open source software relies on individuals who select their own tasks, it is often *underproduced*—a term used by software engineering researchers to describe when a piece of software's relative quality is lower than its relative importance. We examine the social and technical factors associated with underproduction in a population of nearly 21,000 packages in the Debian GNU/Linux community. We test a series of hypotheses developed from a reading of prior research in software engineering. Although we find that software age and programming language age offer a partial explanation for variation in underproduction, we were surprised to find that the association between underproduction and package age is weaker at high levels of programming language age. With respect to maintenance efforts, we find that additional resources are not associated with lower underproduction. Surprisingly, larger numbers of contributors is associated with higher underproduction risk. Also, contrary to our expectations, maintainer turnover and maintenance by a declared team are not associated with lower rates of underproduction. Finally, we find that the people working on bugs in underproduced packages tend to be those who are more central to the community's collaboration network structure, although contributors' betweenness centrality (often associated with brokerage in social networks) is not associated with underproduction.

I. Introduction

Open source software is frequently supported by teams of developers, system engineers, designers, and support specialists. While these teams are often organized as both firms, they increasingly take the form of networks of self-organized collaborators working together in a model called commons-based peer production [1]. Although the results of peer production are often innovative and influential (e.g., GNU/Linux, Apache, and Python), the tasks taken on by contributors in these efforts do not always align with tasks that are most needed by the software project's users or by the general public.

Of particular concern is software that is *underproduced*—i.e., low quality, but highly important. Underproduction has been shown to be widespread in open source software [2]. Some underproduced software may be buried deep in the software supply chain, and vulnerabilities and flaws may not be noticed until they cause disruptions. How can we identify underproduced software and remediate risks before they cause major failures?

This paper builds on previous software engineering research focused on risk measurement to test a series of hypotheses on correlates and theorized causes of underproduction in open source software oriented to both material conditions (e.g., the programming language used) and social features (e.g., the number of maintainers). We open in §II with a review of related work to build intuition around these hypotheses, describe our setting and methods in further detail in §III, and present our analysis in §IV. We discuss the implications of our results in §V, describing limitations around these results in §VI before concluding in §VII.

II. Background

A. The production of Free/Libre Open Source Software

People start and join free/libre open source software (FLOSS) projects for a wide range of reasons, often including intrinsic motivations [3, 4, 5]. The work of these developers can be organized in numerous ways—from individual efforts with few if any other contributors, to casual handovers among whoever is willing to pitch in, to committed small-group collaborations, to networks of thousands of developers coordinating their work and making regular integrated releases [6, 7]. For example, the Apache Project today oversees a widely-used web server but was founded by a group of system administrators who had been informally trading fixes to an older, abandoned piece of software [8]. The Linux kernel was created by Linus Torvalds as a personal project [1] before being shared with the world at no charge. The openness and flexibility of how work is organized do not guarantee participation. Volunteers may trickle in slowly, if at all, or be treated so poorly that they leave [9]. Ultimately, software may come to rely heavily on a few individuals or even a single person. Leaders can burn out and may not have a pipeline of candidates to assume key roles [10, 11].

Although FLOSS's impact can be large, the process that creates these goods can be inefficient. Benkler observed that some of the largest and most well-known FLOSS projects function as commons-based peer production organizations that rely on voluntary and self-organized labor [12]. Although a substantial portion of FLOSS is supported by firms [13], these firms do not generally assign tasks within FLOSS projects. Individuals—especially volunteers—tend to choose their own tasks. Unfortunately, the tasks individuals choose may not be those most needed by users. As a result, underproduction—when the quality of a good falls below its importance—can introduce an important form of risk.

- Age is important (but attenuates)
- Individuals and small teams perform well…
  but this is not always practical.

Takeaways

Sources of Underproduction in Open Source Software

Kaylea Champion
University of Washington
kaylea@uw.edu

Benjamin Mako Hill
University of Washington
makohill@uw.edu

Abstract—Because open source software relies on individuals who select their own tasks, it is often *underproduced*—a term used by software engineering researchers to describe when a piece of software's relative quality is lower than its relative importance. We examine the social and technical factors associated with underproduction in a population of nearly 3 million packages in the Debian GNU/Linux community. We test a series of hypotheses developed from a reading of prior research in software engineering. Although we find that software age and programming language age offer a partial explanation for variation in underproduction, we were surprised to find that the association between underproduction and package age is weaker at high levels of programming language age. With respect to maintenance efforts, we find that additional resources are not associated with lower underproduction. Surprisingly, a greater number of contributors is associated with higher underproduction risk. Also, contrary to our expectations, maintainer turnover and maintenance by a declared team are not associated with lower rates of underproduction. Finally, we find that the people working on bugs in underproduced packages tend to be those who are more central to the community's collaboration network structure, although contributors' betweenness centrality (often associated with brokerage in social networks) is not associated with underproduction.

I. Introduction

Open source software is frequently supported by teams of developers, system engineers, designers, and support specialists. While these teams are often organized as both firms, they increasingly take the form of networks of self-organized collaborators working together in a model called commons-based peer production [1]. Although the results of peer production are often innovative and influential (e.g., GNU/Linux, Apache, and Python), the tasks taken on by contributors in these efforts do not always align with tasks that are most needed by the software project's users or by the general public.

Of particular concern is software that is *underproduced*—i.e., low quality, but highly important. Underproduction has been shown to be widespread in open source software [2]. Some underproduced software may be buried deep in the software supply chain, and vulnerabilities and flaws may not be noticed until they cause disruptions. How can we identify underproduced software and remediate risks before they cause major failures?

This paper builds on previous software engineering research focused on risk measurement to test a series of hypotheses on correlates and theorized causes of underproduction in open source software oriented to both material conditions (e.g., the programming language used) and social features (e.g., the number of maintainers). We open in §II with a review of related work to build intuition around these hypotheses, describe our setting and methods in further detail in §III, and present our analysis in §IV. We discuss the implications of our results in §V, describing limitations around these results in §VI before concluding in §VII.

II. Background

A. The production of Free/Libre Open Source Software

People start and join free/libre open source software (FLOSS) projects for a wide range of reasons, often including intrinsic motivations [3, 4, 5]. The work of these developers can be organized in numerous ways—from individual efforts with few if any other contributors, to casual handovers among whoever is willing to pitch in, to committed small-group collaborations, to networks of thousands of developers coordinating their work and making regular integrated releases [6, 7]. For example, the Apache Project today oversees a widely-used web server but was founded by a group of system administrators who had been informally trading fixes to an older, abandoned piece of software [8]. The Linux kernel was created by Linus Torvalds as a personal project [1] before being shared with the world at no charge. The openness and flexibility of how work is organized do not guarantee participation. Volunteers may trickle in slowly, if at all, or be treated so poorly that they leave [9]. Ultimately, software may come to rely heavily on a few individuals or even a single person. Leaders can burn out and may not have a pipeline of candidates to assume key roles [10, 11].

Although FLOSS's impact can be large, the process that creates these goods can be inefficient. Benkler observed that some of the largest and most well-known FLOSS projects function as commons-based peer production organizations that rely on voluntary and self-organized labor [12]. Although a substantial portion of FLOSS is supported by firms [13], these firms do not generally assign tasks within FLOSS projects. Individuals—especially volunteers—tend to choose their own tasks. Unfortunately, the tasks individuals choose may not be those most needed by users. As a result, underproduction—when the quality of a good falls below its importance—can introduce an important form of risk.

- Age is important (but attenuates)
- Individuals and small teams perform well…
  but this is not always practical.
- Core people are already carrying substantial burden.

Takeaways

Sources of Underproduction in Open Source Software

Kaylea Champion
University of Washington
kaylea@uw.edu

Benjamin Mako Hill
University of Washington
makohill@uw.edu

Abstract—Because open source software relies on individuals who select their own tasks, it is often *underproduced*—a term used by software engineering researchers to describe when a piece of software's relative quality is lower than its relative importance. We examine the social and technical factors associated with underproduction in a population of nearly 3 million packages in the Debian GNU/Linux community. We test a series of hypotheses developed from a reading of prior research in software engineering. Although we find that software age and programming language age offer a partial explanation for variation in underproduction, we were surprised to find that the association between underproduction and package age is weaker at high levels of programming language age. With respect to maintenance efforts, we find that additional resources are not always helpful: packages maintained by larger and more numerous sets of contributors is associated with higher underproduction risk. Also, contrary to our expectations, maintainer turnover and maintenance by a declared team are not associated with lower rates of underproduction. Finally, we find that the people working on bugs in underproduced packages tend to be those who are more central to the community's collaboration network structure, although contributors' betweenness centrality (often associated with brokerage in social networks) is not associated with underproduction.

I. Introduction

Open source software is frequently supported by teams of developers, system engineers, designers, and support specialists. While these teams are often organized as both firms, they increasingly take the form of networks of self-organized collaborators working together in a model called commons-based peer production [1]. Although the results of peer production are often innovative and influential (e.g., GNU/Linux, Apache, and Python), the tasks taken on by contributors in these efforts do not always align with tasks that are most needed by the software project's users or by the general public.

Of particular concern is software that is *underproduced*—i.e., low quality, but highly important. Underproduction has been shown to be widespread in open source software [2]. Some underproduced software may be buried deep in the software supply chain, and vulnerabilities and flaws may not be noticed until they cause disruptions. How can we identify underproduced software and remediate risks before they cause major failures?

This paper builds on previous software engineering research focused on risk measurement to test a series of hypotheses on correlates and theorized causes of underproduction in open source software oriented to both material conditions (e.g., the programming language used) and social features (e.g., the number of maintainers). We open in §II with a review of related work to build intuition around these hypotheses, describe our setting and methods in further detail in §III, and present our analysis in §IV. We discuss the implications of our results in §V, describing limitations around these results in §VI before concluding in §VII.

II. Background

A. The production of Free/Libre Open Source Software

People start and join free/libre open source software (FLOSS) projects for a wide range of reasons, often including intrinsic motivations [3, 4, 5]. The work of these developers can be organized in numerous ways—from individual efforts with few if any other contributors, to casual handovers among whoever is willing to pitch in, to committed small-group collaborations, to networks of thousands of developers coordinating their work and making regular integrated releases [6, 7]. For example, the Apache Project today oversees a widely-used web server but was founded by a group of system administrators who had been informally trading fixes to an older, abandoned piece of software [8]. The Linux kernel was created by Linus Torvalds as a personal project [1] before being shared with the world at no charge. The openness and flexibility of how work is organized do not guarantee participation. Volunteers may trickle in slowly, if at all, or be treated so poorly that they leave [9]. Ultimately, software may come to rely heavily on a few individuals or even a single person. Leaders can burn out and may not have a pipeline of candidates to assume key roles [10, 11].

Although FLOSS's impact can be large, the process that creates these goods can be inefficient. Benkler observed that some of the largest and most well-known FLOSS projects function as commons-based peer production organizations that rely on voluntary and self-organized labor [12]. Although a substantial portion of FLOSS is supported by firms [13], these firms do not generally assign tasks within FLOSS projects. Individuals—especially volunteers—tend to choose their own tasks. Unfortunately, the tasks individuals choose may not be those most needed by users. As a result, underproduction—when the quality of a good falls below its importance—can introduce an important form of risk.

- Age is important (but attenuates)
- Individuals and small teams perform well…
  but this is not always practical.
- Core people are already carrying substantial burden.
- We have more to learn…

Takeaways

Sources of Underproduction in Open Source Software

Kaylea Champion
University of Washington
kaylea@uw.edu

Benjamin Mako Hill
University of Washington
makohill@uw.edu

***Abstract*—Because open source software relies on individuals who select their own tasks, it is often *underproduced*—a term used by software engineering researchers to describe when a piece of software's relative quality is lower than its relative importance. We examine the social and technical factors associated with underproduction through a comparison of software packaged by the Debian GNU/Linux community. We test a series of hypotheses developed from a reading of prior research in software engineering. Although we find that software age and programming language age offer a partial explanation for variation in underproduction, we were surprised to find that the association between underproduction and package age is weaker at high levels of programming language age. With respect to maintenance efforts, we find that additional resources are not always tied to better outcomes. In particular, having higher numbers of contributors is associated with higher underproduction risk. Also, contrary to our expectations, maintainer turnover and maintenance by a declared team are not associated with lower rates of underproduction. Finally, we find that the people working on bugs in underproduced packages tend to be those who are more central to the community's collaboration network structure, although contributors' betweenness centrality (often associated with brokerage in social networks) is not associated with underproduction.**

I. Introduction

Open source software is frequently supported by teams of developers, system engineers, designers, and support specialists. While these teams are often organized as both firms, they increasingly take the form of networks of self-organized collaborators working together in a model called commons-based peer production [1]. Although the results of peer production are often innovative and influential (e.g., GNU/Linux, Apache, and Python), the tasks taken on by contributors in these efforts do not always align with tasks that are most needed by the software project's users or by the general public.

Of particular concern is software that is *underproduced*—i.e., low quality, but highly important. Underproduction has been shown to be widespread in open source software [2]. Some underproduced software may be buried deep in the software supply chain, and vulnerabilities and flaws may not be noticed until they cause disruptions. How can we identify underproduced software and remediate risks before they cause major failures?

This paper builds on previous software engineering research focused on risk measurement to test a series of hypotheses on correlates and theorized causes of underproduction in open source software oriented to both material conditions (e.g., the programming language used) and social features (e.g., the number of maintainers). We open in §II with a review of related work to build intuition around these hypotheses, describe our setting and methods in further detail in §III, and present our analysis in §IV. We discuss the implications of our results in §V, describing limitations around these results in §VI before concluding in §VII.

II. Background

A. The production of Free/Libre Open Source Software

People start and join free/libre open source software (FLOSS) projects for a wide range of reasons, often including intrinsic motivations [3, 4, 5]. The work of these developers can be organized in numerous ways—from individual efforts with few if any other contributors, to casual handovers among whoever is willing to pitch in, to committed small-group collaborations, to networks of thousands of developers coordinating their work and making regular integrated releases [6, 7]. For example, the Apache Project today oversees a widely-used web server but was founded by a group of system administrators who had been informally trading fixes to an older, abandoned piece of software [8]. The Linux kernel was created by Linus Torvalds as a personal project [1] before being shared with the world at no charge. The openness and flexibility of how work is organized do not guarantee participation. Volunteers may trickle in slowly, if at all, or be treated so poorly that they leave [9]. Ultimately, software may come to rely heavily on a few individuals or even a single person. Leaders can burn out and may not have a pipeline of candidates to assume key roles [10, 11].

Although FLOSS's impact can be large, the process that creates these goods can be inefficient. Benkler observed that some of the largest and most well-known FLOSS projects function as commons-based peer production organizations that rely on voluntary and self-organized labor [12]. Although a substantial portion of FLOSS is supported by firms [13], these firms do not generally assign tasks within FLOSS projects. Individuals—especially volunteers—tend to choose their own tasks. Unfortunately, the tasks individuals choose may not be those most needed by users. As a result, underproduction—when the quality of a good falls below its importance—can introduce an important form of risk.

- Age is important (but attenuates)
- Individuals and small teams perform well…
  but this is not always practical.
- Core people are already carrying substantial burden.
- We have more to learn…
  about carrying capacity,

Takeaways

Sources of Underproduction in Open Source Software

Kaylea Champion
University of Washington
kaylea@uw.edu

Benjamin Mako Hill
University of Washington
makohill@uw.edu

Abstract—Because open source software relies on individuals who select their own tasks, it is often *underproduced*—a term used by software engineering researchers to describe when a piece of software's relative quality is lower than its relative importance. We examine the social and technical factors associated with underproduction in a population of nearly 21,000 packages in the Debian GNU/Linux community. We test a series of hypotheses developed from a reading of prior research in software engineering. Although we find that software age and programming language age offer a partial explanation for variation in underproduction, we were surprised to find that the association between underproduction and package age is weaker at high levels of programming language age. With respect to maintenance efforts, we find that additional resources are not associated with lower underproduction. Similarly, a greater number of contributors is associated with higher underproduction risk. Also, contrary to our expectations, maintainer turnover and maintenance by a declared team are not associated with lower rates of underproduction. Finally, we find that the people working on bugs in underproduced packages tend to be those who are more central to the community's collaboration network structure, although contributors' betweenness centrality (often associated with brokerage in social networks) is not associated with underproduction.

I. Introduction

Open source software is frequently supported by teams of developers, system engineers, designers, and support specialists. While these teams are often organized as both firms, they increasingly take the form of networks of self-organized collaborators working together in a model called commons-based peer production [1]. Although the results of peer production are often innovative and influential (e.g., GNU/Linux, Apache, and Python), the tasks taken on by contributors in these efforts do not always align with tasks that are most needed by the software project's users or by the general public.

Of particular concern is software that is *underproduced*—i.e., low quality, but highly important. Underproduction has been shown to be widespread in open source software [2]. Some underproduced software may be buried deep in the software supply chain, and vulnerabilities and flaws may not be noticed until they cause disruptions. How can we identify underproduced software and remediate risks before they cause major failures?

This paper builds on previous software engineering research focused on risk measurement to test a series of hypotheses on correlates and theorized causes of underproduction in open source software oriented to both material conditions (e.g., the programming language used) and social features (e.g., the number of maintainers). We open in §II with a review of related work to build intuition around these hypotheses, describe our setting and methods in further detail in §III, and present our analysis in §IV. We discuss the implications of our results in §V, describing limitations around these results in §VI before concluding in §VII.

II. Background

A. The production of Free/Libre Open Source Software

People start and join free/libre open source software (FLOSS) projects for a wide range of reasons, often including intrinsic motivations [3, 4, 5]. The work of these developers can be organized in numerous ways—from individual efforts with few if any other contributors, to casual handovers among whoever is willing to pitch in, to committed small-group collaborations, to networks of thousands of developers coordinating their work and making regular integrated releases [6, 7]. For example, the Apache Project today oversees a widely-used web server but was founded by a group of system administrators who had been informally trading fixes to an older, abandoned piece of software [8]. The Linux kernel was created by Linus Torvalds as a personal project [1] before being shared with the world at no charge. Low openness and flexibility of how work is organized do not guarantee participation. Volunteers may trickle in slowly, if at all, or be treated so poorly that they leave [9]. Ultimately, software may come to rely heavily on a few individuals or even a single person. Leaders can burn out and may not have a pipeline of candidates to assume key roles [10, 11].

Although FLOSS's impact can be large, the process that creates these goods can be inefficient. Benkler observed that some of the largest and most well-known FLOSS projects function as commons-based peer production organizations that rely on voluntary and self-organized labor [12]. Although a substantial portion of FLOSS is supported by firms [13], these firms do not generally assign tasks within FLOSS projects. Individuals—especially volunteers—tend to choose their own tasks. Unfortunately, the tasks individuals choose may not be those most needed by users. As a result, underproduction—when the quality of a good falls below its importance—can introduce an important form of risk.

- Age is important (but attenuates)
- Individuals and small teams perform well…
  but this is not always practical.
- Core people are already carrying substantial burden.
- We have more to learn…
  about carrying capacity,
  setting priorities,

Takeaways

Sources of Underproduction in Open Source Software

Kaylea Champion
University of Washington
kaylea@uw.edu

Benjamin Mako Hill
University of Washington
makohill@uw.edu

***Abstract*—Because open source software relies on individuals who select their own tasks, it is often *underproduced*—a term used by software engineering researchers to describe when a piece of software's relative quality is lower than its relative importance. We examine the social and technical factors associated with underproduction through a comparison of software packaged by the Debian GNU/Linux community. We test a series of hypotheses developed from a reading of prior research in software engineering. Although we find that software age and programming language age offer a partial explanation for variation in underproduction, we were surprised to find that the association between underproduction and package age is weaker at high levels of programming language age. With respect to maintenance efforts, we find that additional resources are not always tied to better outcomes. In particular, having higher numbers of contributors is associated with higher underproduction risk. Also, contrary to our expectations, maintainer turnover and maintenance by a declared team are not associated with lower rates of underproduction. Finally, we find that the people working on bugs in underproduced packages tend to be those who are more central to the community's collaboration network structure, although contributors' betweenness centrality (often associated with brokerage in social networks) is not associated with underproduction.**

I. INTRODUCTION

Open source software is frequently supported by teams of developers, system engineers, designers, and support specialists. While these teams are often organized as both firms, they increasingly take the form of networks of self-organized collaborators working together in a model called commons-based peer production [1]. Although the results of peer production are often innovative and influential (e.g., GNU/Linux, Apache, and Python), the tasks taken on by contributors in these efforts do not always align with tasks that are most needed by the software project's users or by the general public.

Of particular concern is software that is *underproduced*—i.e., low quality, but highly important. Underproduction has been shown to be widespread in open source software [2]. Some underproduced software may be buried deep in the software supply chain, and vulnerabilities and flaws may not be noticed until they cause disruptions. How can we identify underproduced software and remediate risks before they cause major failures?

This paper builds on previous software engineering research focused on risk measurement to test a series of hypotheses on correlates and theorized causes of underproduction in open source software oriented to both material conditions (e.g., the programming language used) and social features (e.g., the number of maintainers). We open in §II with a review of related work to build intuition around these hypotheses, describe our setting and methods in further detail in §III, and present our analysis in §IV. We discuss the implications of our results in §V, describing limitations around these results in §VI before concluding in §VII.

II. BACKGROUND

A. The production of Free/Libre Open Source Software

People start and join free/libre open source software (FLOSS) projects for a wide range of reasons, often including intrinsic motivations [3, 4, 5]. The work of these developers can be organized in numerous ways—from individual efforts with few if any other contributors, to casual handovers among whoever is willing to pitch in, to committed small-group collaborations, to networks of thousands of developers coordinating their work and making regular integrated releases [6, 7]. For example, the Apache Project today oversees a widely-used web server but was founded by a group of system administrators who had been informally trading fixes to an older, abandoned piece of software [8]. The Linux kernel was created by Linus Torvalds as a personal project [1] before being shared with the world at no charge. The openness and flexibility of how work is organized do not guarantee participation. Volunteers may trickle in slowly, if at all, or be treated so poorly that they leave [9]. Ultimately, software may come to rely heavily on a few individuals or even a single person. Leaders can burn out and may not have a pipeline of candidates to assume key roles [10, 11].

Although FLOSS's impact can be large, the process that creates these goods can be inefficient. Benkler observed that some of the largest and most well-known FLOSS projects function as commons-based peer production organizations that rely on voluntary and self-organized labor [12]. Although a substantial portion of FLOSS is supported by firms [13], these firms do not generally assign tasks within FLOSS projects. Individuals—especially volunteers—tend to choose their own tasks. Unfortunately, the tasks individuals choose may not be those most needed by users. As a result, underproduction—when the quality of a good falls below its importance—can introduce an important form of risk.

- Age is important (but attenuates)
- Individuals and small teams perform well…

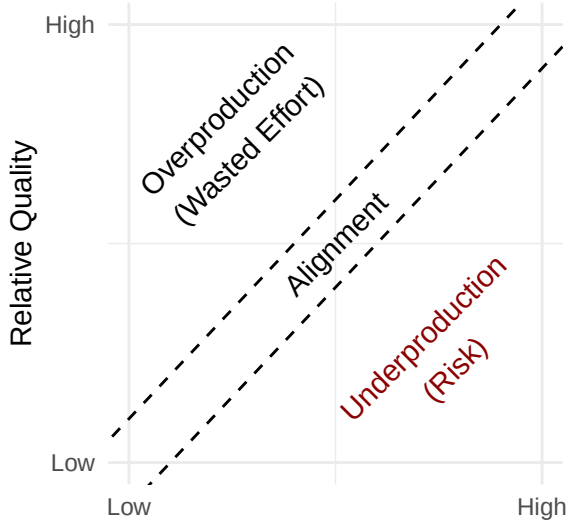
  but this is not always practical.
- Core people are already carrying substantial burden.
- We have more to learn…

  about carrying capacity,

  setting priorities,

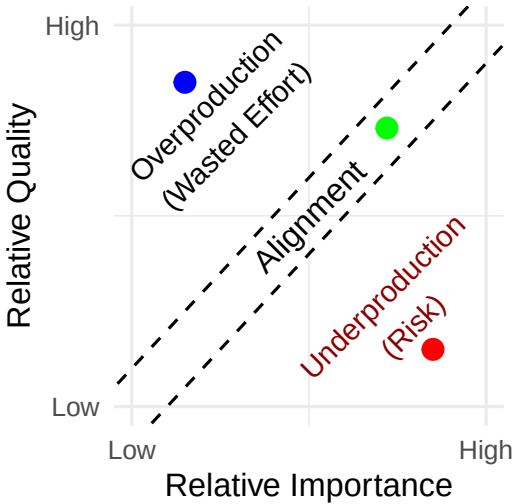
  and development team structure.

Recasting Cross-Section to Longitudinal Analysis



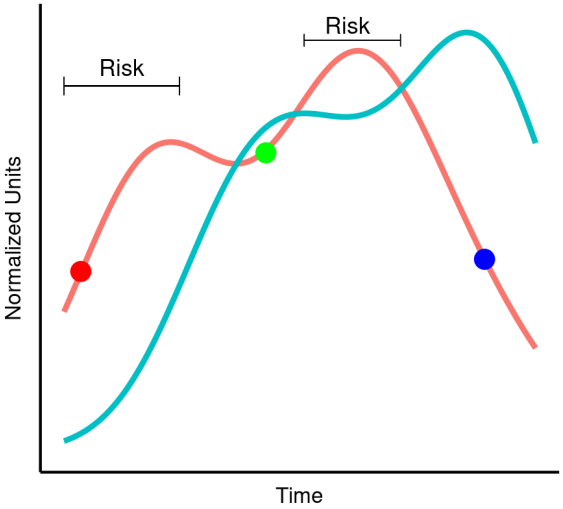
[Champion, K., and Hill, B. M. (2025, Working Paper)]

Recasting Cross-Section to Longitudinal Analysis



[Champion, K., and Hill, B. M. (2025, Working Paper)]

Taking a Lifecycle Perspective



[Champion, K., and Hill, B. M. (2025, Working Paper)]

Dynamic Underproduction Analysis

1. Identify a longitudinal measure of importance.

[Champion, K., and Hill, B. M. (2025, Working Paper)]

Dynamic Underproduction Analysis

1. Identify a longitudinal measure of importance.
2. Identify a longitudinal measure of quality or security.

[Champion, K., and Hill, B. M. (2025, Working Paper)]

Dynamic Underproduction Analysis

1. Identify a longitudinal measure of importance.
2. Identify a longitudinal measure of quality or security.
3. Make units comparable.

[Champion, K., and Hill, B. M. (2025, Working Paper)]

Dynamic Underproduction Analysis

1. Identify a longitudinal measure of importance.
2. Identify a longitudinal measure of quality or security.
3. Make units comparable.
4. Propose a preferable temporal relationship.

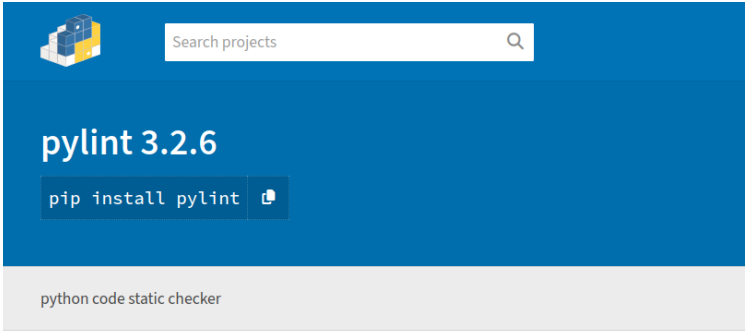
[Champion, K., and Hill, B. M. (2025, Working Paper)]

Dynamic Underproduction Analysis

1. Identify a longitudinal measure of importance.
2. Identify a longitudinal measure of quality or security.
3. Make units comparable.
4. Propose a preferable temporal relationship.
5. Extract relevant comparisons.

[Champion, K., and Hill, B. M. (2025, Working Paper)]

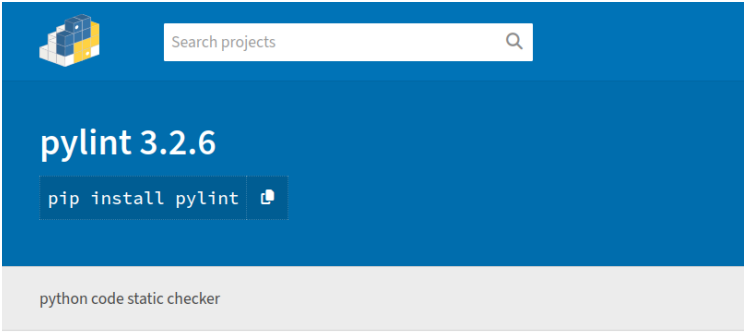
Measuring Quality Longitudinally



- Stepping upstream in the supply chain

[Champion, K., and Hill, B. M. (2025, Working Paper)]

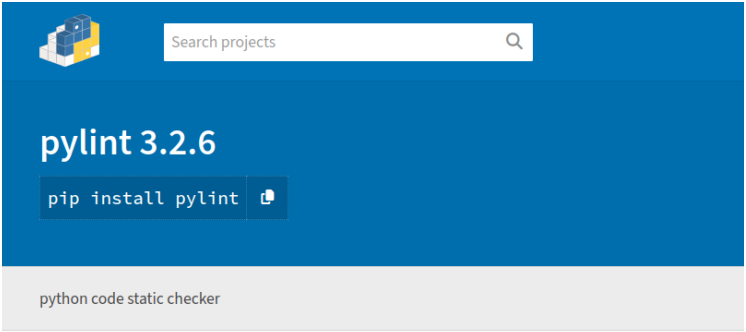
Measuring Quality Longitudinally



- Stepping upstream in the supply chain
- Static analysis using validated measures via tools available to practitioners

[Champion, K., and Hill, B. M. (2025, Working Paper)]

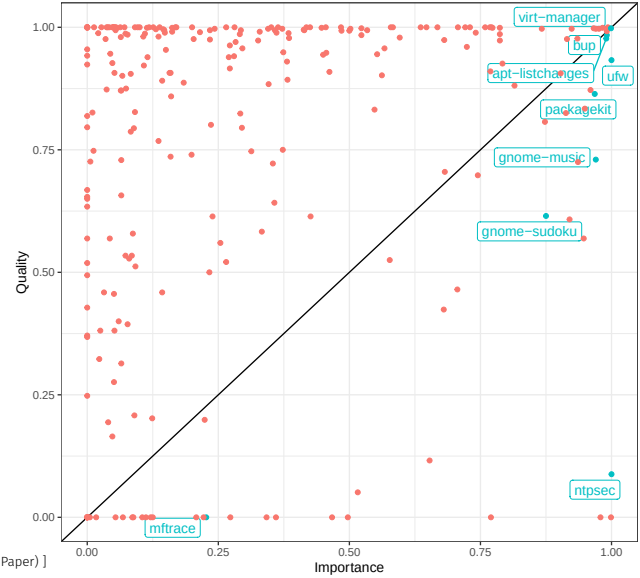
Measuring Quality Longitudinally



- Stepping upstream in the supply chain
- Static analysis using validated measures via tools available to practitioners
- Using Pylint for Python; evaluating Flawfinder and Clang Scan-build for C/C++

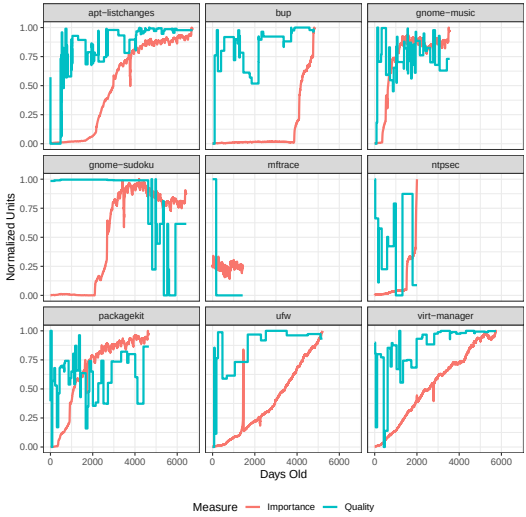
[Champion, K., and Hill, B. M. (2025, Working Paper)]

Taking a Lifecycle Perspective



[Champion, K., and Hill, B. M. (2025, Working Paper)]

Taking a Lifecycle Perspective



[Champion, K., and Hill, B. M. (2025, Working Paper)]

Interventions: What's on the table?

- Development communities and governance

Interventions: What's on the table?

- Development communities and governance
  - Evidence-based decisionmaking
  - Shared power (see Gaughan, Champion & Hwang, *SANER 2024*)
  - Retaining contributors (see Champion & Hill, *New Media and Society 2024*)
  - Minimize toxicity

Interventions: What's on the table?

- Development communities and governance
  - Evidence-based decisionmaking
  - Shared power (see Gaughan, Champion & Hwang, *SANER 2024*)
  - Retaining contributors (see Champion & Hill, *New Media and Society 2024*)
  - Minimize toxicity
- Dissemination, tools, data

Interventions: What's on the table?

- Development communities and governance
  - Evidence-based decisionmaking
  - Shared power (see Gaughan, Champion & Hwang, *SANER 2024*)
  - Retaining contributors (see Champion & Hill, *New Media and Society 2024*)
  - Minimize toxicity
- Dissemination, tools, data
  - Tools to surface metrics for practitioner and public use
  - Sharing of research results
  - Robust methods

Interventions: What's on the table?

- Development communities and governance
  - Evidence-based decisionmaking
  - Shared power (see Gaughan, Champion & Hwang, *SANER 2024*)
  - Retaining contributors (see Champion & Hill, *New Media and Society 2024*)
  - Minimize toxicity
- Dissemination, tools, data
  - Tools to surface metrics for practitioner and public use
  - Sharing of research results
  - Robust methods
- Policies, regulations, & governance

Interventions: What's on the table?

- Development communities and governance
  - Evidence-based decisionmaking
  - Shared power (see Gaughan, Champion & Hwang, *SANER 2024*)
  - Retaining contributors (see Champion & Hill, *New Media and Society 2024*)
  - Minimize toxicity
- Dissemination, tools, data
  - Tools to surface metrics for practitioner and public use
  - Sharing of research results
  - Robust methods
- Policies, regulations, & governance
  - Standards bodies (IEEE, ISO)
  - NIST, CISA
  - EU Cyber Resilience Act (CRA), German Sovereign Technology Agency, United Nations

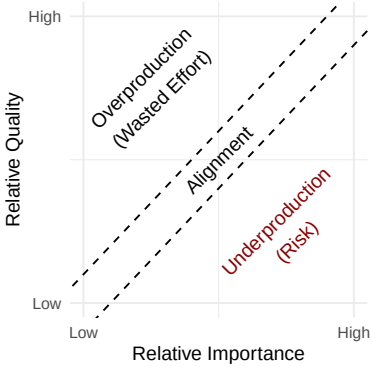
Interventions: What's on the table?

- Development communities and governance
  - Evidence-based decisionmaking
  - Shared power (see Gaughan, Champion & Hwang, *SANER 2024*)
  - Retaining contributors (see Champion & Hill, *New Media and Society 2024*)
  - Minimize toxicity
- Dissemination, tools, data
  - Tools to surface metrics for practitioner and public use
  - Sharing of research results
  - Robust methods
- Policies, regulations, & governance
  - Standards bodies (IEEE, ISO)
  - NIST, CISA
  - EU Cyber Resilience Act (CRA), German Sovereign Technology Agency, United Nations
- Firms & Organizations

Interventions: What's on the table?

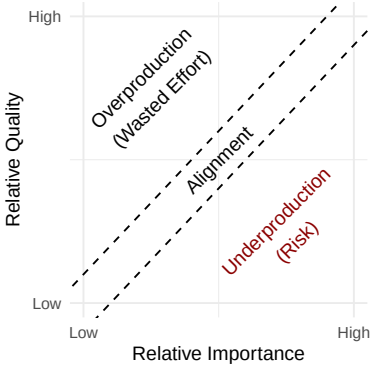
- Development communities and governance
  - Evidence-based decisionmaking
  - Shared power (see Gaughan, Champion & Hwang, *SANER 2024*)
  - Retaining contributors (see Champion & Hill, *New Media and Society 2024*)
  - Minimize toxicity
- Dissemination, tools, data
  - Tools to surface metrics for practitioner and public use
  - Sharing of research results
  - Robust methods
- Policies, regulations, & governance
  - Standards bodies (IEEE, ISO)
  - NIST, CISA
  - EU Cyber Resilience Act (CRA), German Sovereign Technology Agency, United Nations
- Firms & Organizations
  - Awareness and assessment
  - Cooperation and taking responsibility
  - Investment and support

Next areas of work



Current work:

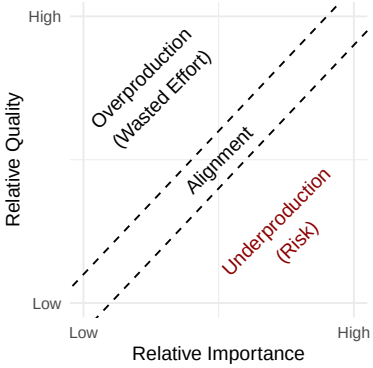
Next areas of work



Current work:

- examine how underproduction emerges over time

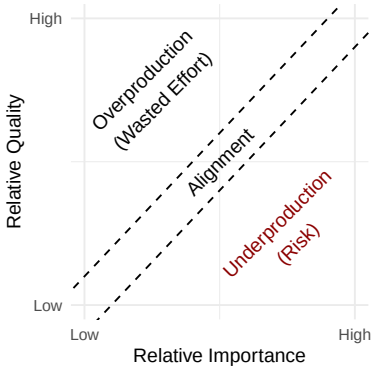
Next areas of work



Current work:

- examine how underproduction emerges over time
- spillover effects (including rise of attention to AI)

Next areas of work

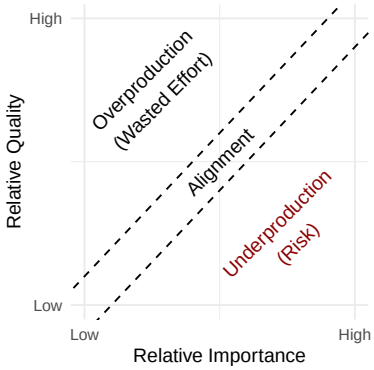


Current work:

- examine how underproduction emerges over time
- spillover effects (including rise of attention to AI)

Near term targets:

Next areas of work



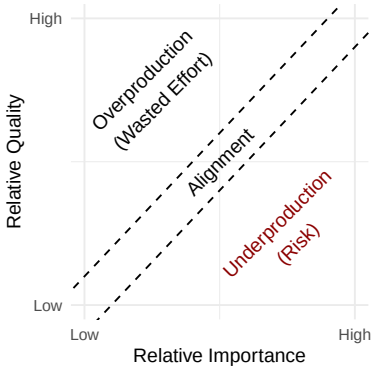
Current work:

- examine how underproduction emerges over time
- spillover effects (including rise of attention to AI)

Near term targets:

- expand measures (esp. cybersecurity)

Next areas of work



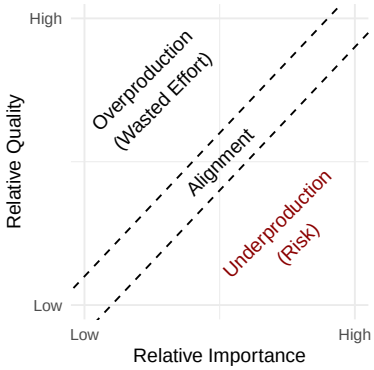
Current work:

- examine how underproduction emerges over time
- spillover effects (including rise of attention to AI)

Near term targets:

- expand measures (esp. cybersecurity)
- expand contexts (new platforms, stacks, languages)

Next areas of work



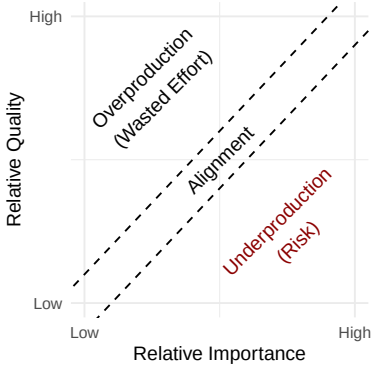
Current work:

- examine how underproduction emerges over time
- spillover effects (including rise of attention to AI)

Near term targets:

- expand measures (esp. cybersecurity)
- expand contexts (new platforms, stacks, languages)
- maintenance transition planning (what can we stop doing?)

Next areas of work



Current work:

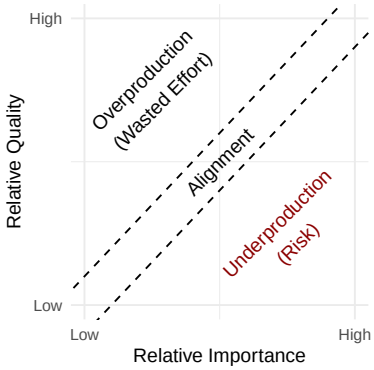
- examine how underproduction emerges over time
- spillover effects (including rise of attention to AI)

Near term targets:

- expand measures (esp. cybersecurity)
- expand contexts (new platforms, stacks, languages)
- maintenance transition planning (what can we stop doing?)

Longer term targets:

Next areas of work



Current work:

- examine how underproduction emerges over time
- spillover effects (including rise of attention to AI)

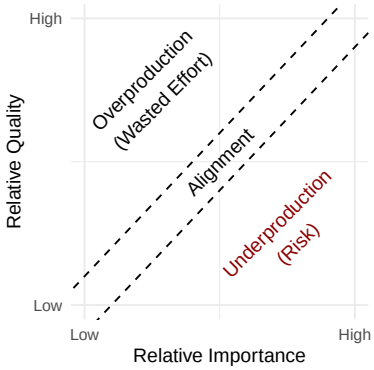
Near term targets:

- expand measures (esp. cybersecurity)
- expand contexts (new platforms, stacks, languages)
- maintenance transition planning (what can we stop doing?)

Longer term targets:

- design, deploy, validate interventions

Next areas of work



Current work:

- examine how underproduction emerges over time
- spillover effects (including rise of attention to AI)

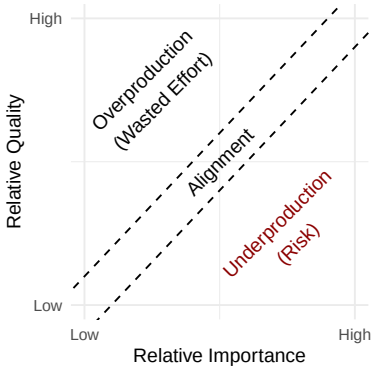
Near term targets:

- expand measures (esp. cybersecurity)
- expand contexts (new platforms, stacks, languages)
- maintenance transition planning (what can we stop doing?)

Longer term targets:

- design, deploy, validate interventions
- underproduction in other domains (scientific software and data, protocols, standards)

Next areas of work



Current work:

- examine how underproduction emerges over time
- spillover effects (including rise of attention to AI)

Near term targets:

- expand measures (esp. cybersecurity)
- expand contexts (new platforms, stacks, languages)
- maintenance transition planning (what can we stop doing?)

Longer term targets:

- design, deploy, validate interventions
- underproduction in other domains (scientific software and data, protocols, standards)
- development lifecycles and strategies, organizational models, and ecosystem design

Thank you!

kaylea@uw.edu—@kaylea@social.coop

`https://kayleachampion.com`

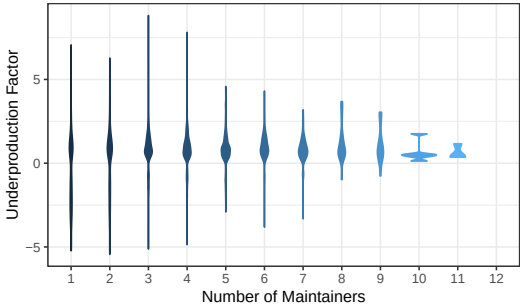
Actively seeking new funding and collaborators!

We gratefully acknowledge support from the Sloan Foundation through the Ford/Sloan Digital Infrastructure Initiative, Sloan Award 2018-11356 as well as the National Science Foundation (Grant IIS-2045055). This work was conducted using the Hyak supercomputer at the University of Washington as well as research computing resources at Northwestern University.

[Molalla River Recreation Area. Bureau of Land Management (blmoregon) via flickr, CC BY 2.0]

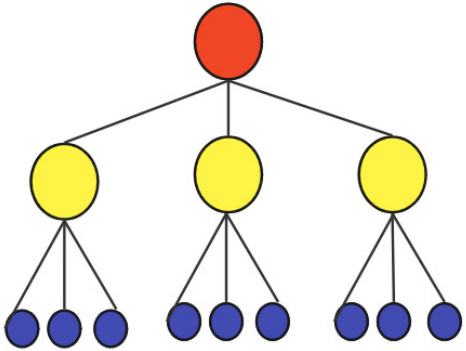
[Washington Coast. John Murphy. (kingair42) via flickr, CC BY-SA 2.0]

Maintainers



Visualizing the optimum number of maintainers for a package.

Betweenness



Brokerage (betweenness centrality) not significant in full model.

[Champion and Hill (SANER 2024)]

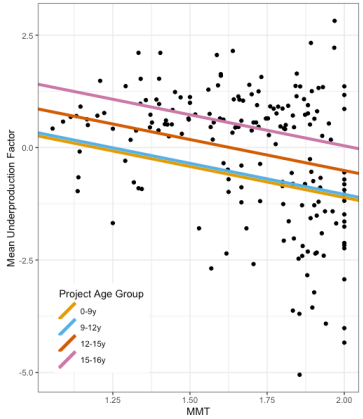
Sources of underprod - full model fit

Table 2: Underproduction is a function of social and technical factors. Coefficients are log-odds estimates with a 95% CI; the number of observations varies due to missing data.

	M1: no lang/network measures	M2: No language measures	M3: No network measures	M4: Full model
(Intercept)	−1.90*	−1.66*	−6.57*	−7.28*
	[−2.07; −1.73]	[−1.91; −1.41]	[−8.24; −4.89]	[−9.06; −5.50]
Package Age (years)	0.14*	0.08*	0.32*	0.34*
	[0.13; 0.15]	[0.06; 0.09]	[0.22; 0.43]	[0.23; 0.45]
Uploader Count	0.21*	0.13*	0.26*	0.18*
	[0.17; 0.24]	[0.09; 0.17]	[0.21; 0.32]	[0.12; 0.24]
Did maintainer change?	0.32*	0.35*	0.27*	0.22
	[0.19; 0.45]	[0.19; 0.51]	[0.03; 0.51]	[−0.03; 0.47]
Team proportion	0.17*	0.03	−0.48*	−0.20
	[0.02; 0.33]	[−0.17; 0.23]	[−0.79; −0.16]	[−0.54; 0.13]
Eigenvector Centrality		16.74*		18.91*
		[13.08; 20.41]		[14.18; 23.64]
Betweenness Centrality		−0.00		−0.00
		[−0.00; 0.00]		[−0.00; 0.00]
Mean Language Age			0.15*	0.16*
			[0.11; 0.19]	[0.12; 0.20]
Package Age : Mean Language Age			−0.01*	−0.01*
			[−0.01; −0.00]	[−0.01; −0.01]
AIC	6586.11	4373.82	2305.08	2088.71
BIC	6619.97	4418.64	2345.35	2140.37
Log Likelihood	−3288.05	−2179.91	−1145.54	−1035.35
Deviance	6576.11	4359.82	2291.08	2070.71
Num. obs.	6450	4459	2328	2299

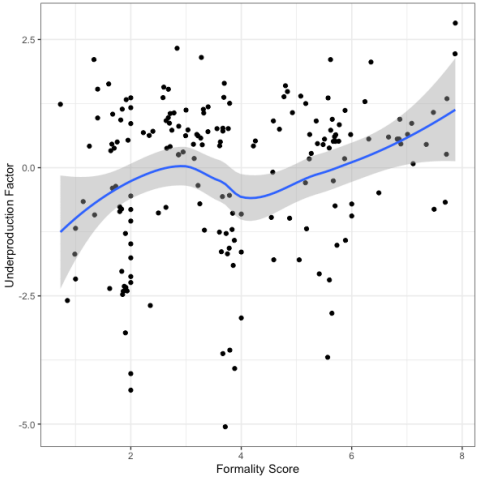
* Null hypothesis value outside the confidence interval.

Power-sharing and Underproduction



Gaughan, M., Champion, K.; and Hwang, S. (2024) Engineering Formality and Software Risk in Debian. *IEEE SANER*.

Engineering formality and Underproduction



Gaughan, M., Champion, K.; and Hwang, S. (2024) Engineering Formality and Software Risk in Debian. *IEEE SANER*.

Retaining Contributors and Underproduction

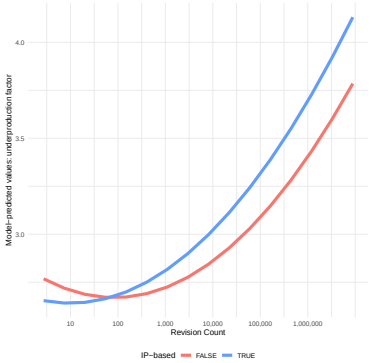


Figure 2: The marginal effect of having higher experience on the average alignment of an article selected for editing. Increasing values indicate increased levels of underproduction, i.e. low-quality but highly-viewed topics.

Champion, K., and Hill, B. M. (2024) Countering underproduction of peer produced goods. *New Media & Society*

Retaining Contributors and Underproduction

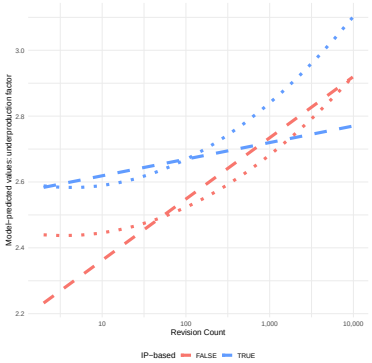
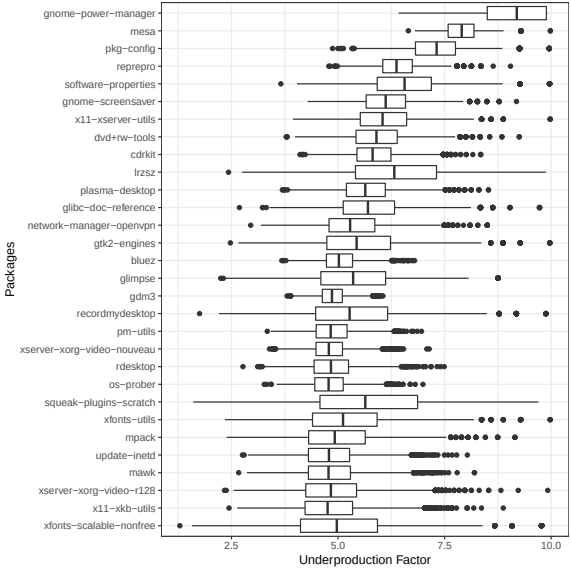


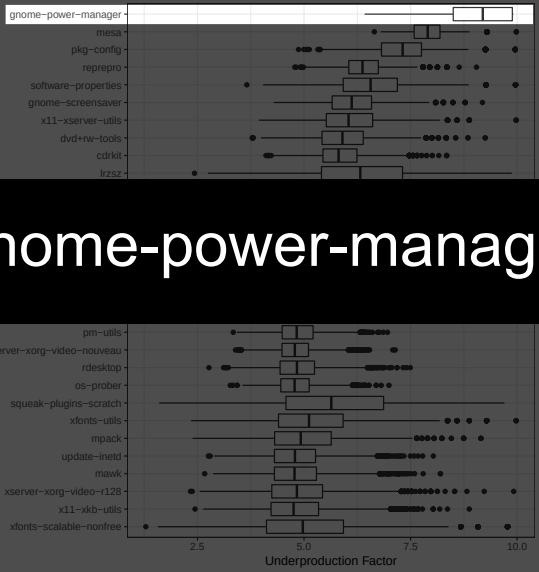
Figure 3: Marginal effect of increased experience on contributor task selection from our within-person sample. We use median individual-level fixed effects. Dashed lines are predicted values using a linear model, while dotted lines use a quadratic model.

Champion, K., and Hill, B. M. (2024) Countering underproduction of peer produced goods. *New Media & Society*

Findings: Underproduction in Debian

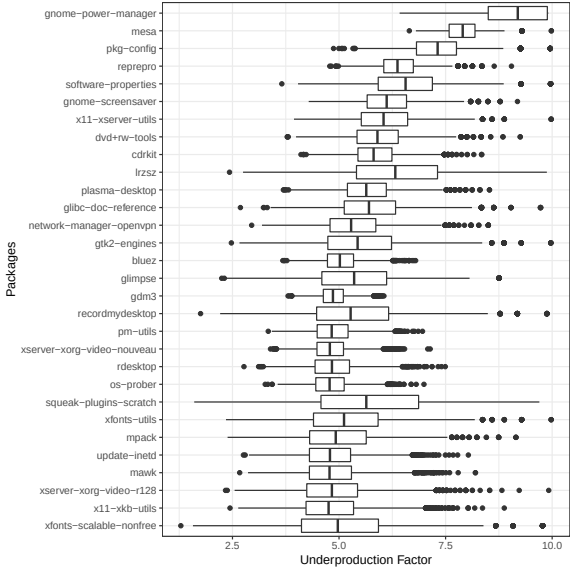


Findings: Underproduction in Debian

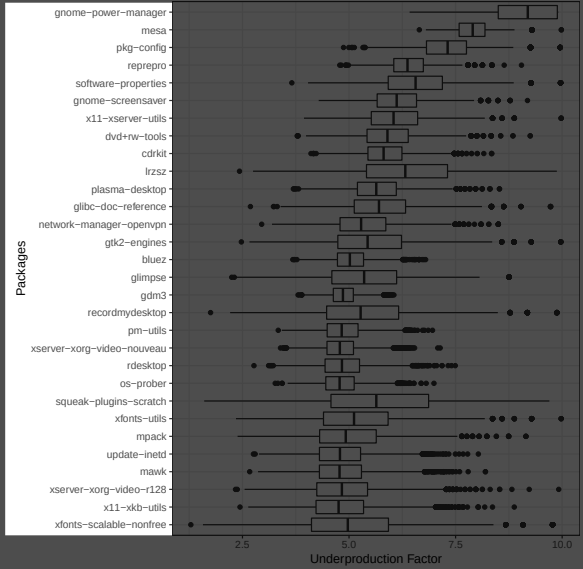


gnome-power-manager

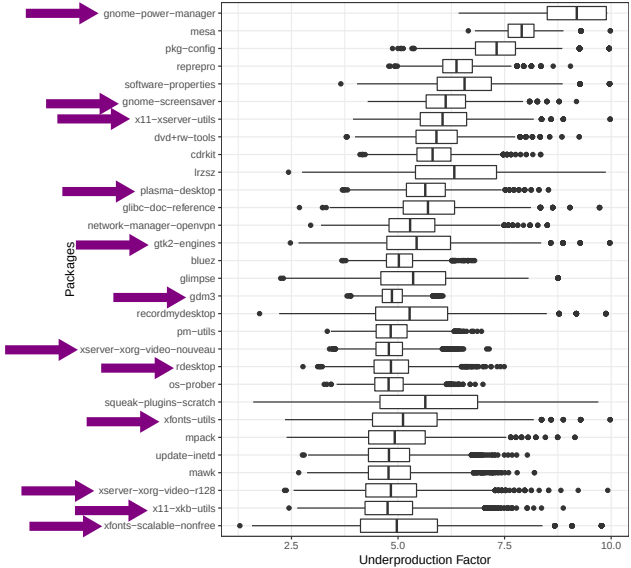
The long tail of GUI underproduction



The long tail of GUI underproduction



The long tail of GUI underproduction



Commons-based peer production is characterized by:

- self-selection of tasks

Benkler, Y. (2006) *Wealth of Networks*

Commons-based peer production is characterized by:

- self-selection of tasks
- sharing, collaboration

Benkler, Y. (2006) *Wealth of Networks*

Commons-based peer production is characterized by:

- self-selection of tasks
- sharing, collaboration
- combining modular work

Benkler, Y. (2006) *Wealth of Networks*

Commons-based peer production is characterized by:

- self-selection of tasks
- sharing, collaboration
- combining modular work
- diverse motivations

Benkler, Y. (2006) *Wealth of Networks*

Commons-based peer production is characterized by:

- self-selection of tasks
- sharing, collaboration
- combining modular work
- diverse motivations
  - fun, kindness, reciprocity

Benkler, Y. (2006) *Wealth of Networks*

Commons-based peer production is characterized by:

- self-selection of tasks
- sharing, collaboration
- combining modular work
- diverse motivations
  - fun, kindness, reciprocity
  - convenience, professional standards

Benkler, Y. (2006) *Wealth of Networks*

Commons-based peer production is characterized by:

- self-selection of tasks
- sharing, collaboration
- combining modular work
- diverse motivations
  - fun, kindness, reciprocity
  - convenience, professional standards
  - paid and volunteer labor

Benkler, Y. (2006) *Wealth of Networks*

Commons-based peer production is characterized by:

- self-selection of tasks
- sharing, collaboration
- combining modular work
- diverse motivations
  - fun, kindness, reciprocity
  - convenience, professional standards
  - paid and volunteer labor
- diverse goals

Benkler, Y. (2006) *Wealth of Networks*

Commons-based peer production is characterized by:

- self-selection of tasks
- sharing, collaboration
- combining modular work
- diverse motivations
  - fun, kindness, reciprocity
  - convenience, professional standards
  - paid and volunteer labor
- diverse goals
  - local problem solving

Benkler, Y. (2006) *Wealth of Networks*

Commons-based peer production is characterized by:

- self-selection of tasks
- sharing, collaboration
- combining modular work
- diverse motivations
  - fun, kindness, reciprocity
  - convenience, professional standards
  - paid and volunteer labor
- diverse goals
  - local problem solving
  - software development strategy

Benkler, Y. (2006) *Wealth of Networks*

Commons-based peer production is characterized by:

- self-selection of tasks
- sharing, collaboration
- combining modular work
- diverse motivations
  - fun, kindness, reciprocity
  - convenience, professional standards
  - paid and volunteer labor
- diverse goals
  - local problem solving
  - software development strategy
  - business model

Benkler, Y. (2006) *Wealth of Networks*

Information public goods?

- Public goods

Information public goods?

- Public goods
  - Non-excludable

Information public goods?

- Public goods
  - Non-excludable
  - Non-rivalrous

Information public goods?

- Public goods
  - Non-excludable
  - Non-rivalrous
  - Classic examples: clean air, public parks, national defense

Information public goods?

- Public goods
  - Non-excludable
  - Non-rivalrous
  - Classic examples: clean air, public parks, national defense
- Information public goods

Information public goods?

- Public goods
  - Non-excludable
  - Non-rivalrous
  - Classic examples: clean air, public parks, national defense
- Information public goods
  - Security / privacy

Information public goods?

- Public goods
  - Non-excludable
  - Non-rivalrous
  - Classic examples: clean air, public parks, national defense
- Information public goods
  - Security / privacy
  - Knowledge

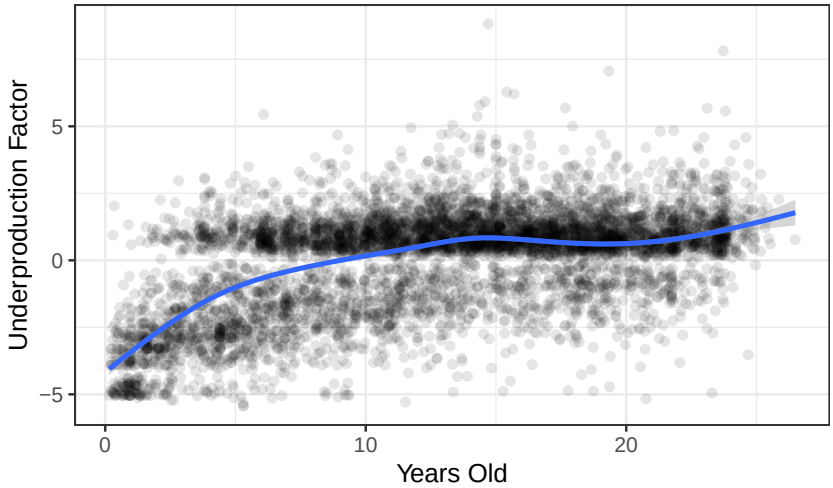
Information public goods?

- Public goods
  - Non-excludable
  - Non-rivalrous
  - Classic examples: clean air, public parks, national defense
- Information public goods
  - Security / privacy
  - Knowledge
  - Protocols and standards

Information public goods?

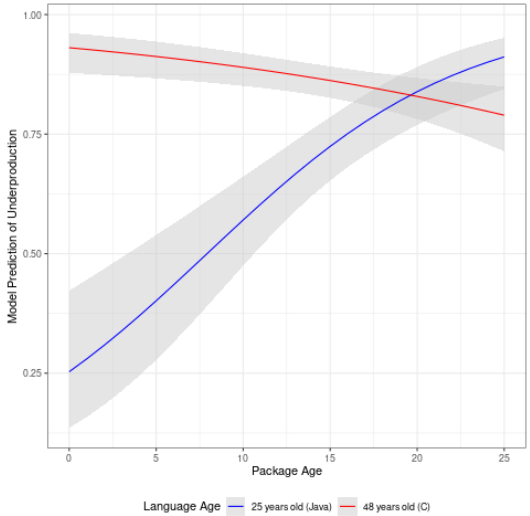
- Public goods
  - Non-excludable
  - Non-rivalrous
  - Classic examples: clean air, public parks, national defense
- Information public goods
  - Security / privacy
  - Knowledge
  - Protocols and standards
  - Open source software

Package Age



[Champion and Hill (SANER 2024)]

Package and Language Age (Interaction Term)



[Champion and Hill (SANER 2024)]

Marginal effects: visualizing x on y for some fixed values of z_1, z_2, etc.

Underproduction: when supply is insufficient to meet demand

[Market. Andrew Gwozdziewycz (thinkiandy) via flickr, CC-BY-NC-ND 2.0]

…but the ‘supply’ of new copies of software is practically infinite, and the price of this software is often ‘free’….

Where might alignment of quality/security and importance come from?

When digital infrastructure is at its best,

- users and producers respond to one another

Where might alignment of quality/security and importance come from?

When digital infrastructure is at its best,

- users and producers respond to one another
- such that the most important software is the best quality and the most secure.