A Theory of Open Source Security: The Spillover of Security Knowledge in Vulnerability Disclosures through Software Supply Chains

Yu-Kai Lin
yklin@gsu.edu
Georgia State University

Weifeng Li
weifeng.li@uga.edu
University of Georgia

Abstract

Open source software (OSS) is critical to digital sovereignty and is required for the modern digital economy. Despite being widely used and highly valued, OSS is not free from security defects. Recent discoveries of critical vulnerabilities in OSS, such as "Log4Shell" and "Heartbleed," underscore the importance of, and lack of theories about, *open source security*. Drawing on organizational learning theory and viewing OSS from the perspective of software supply chains, this study offers a novel theoretical perspective into positive knowledge spillover of vulnerability disclosures in the OSS ecosystem. This occurs when an OSS project (that is, a supplier) discloses a software vulnerability. The security knowledge will be transferred through software supply chains to downstream OSS projects (i.e., consumers), enabling the latter group to better identify new vulnerabilities with similar technical weaknesses in their own code repositories. We further theorized that the severity of the supplier's vulnerability moderates knowledge spillover, where a critical vulnerability, as compared to a noncritical one, yields a much higher spillover that induces interorganizational learning. To validate our theoretical predictions, we conducted a comprehensive analysis using data assembled from the National Vulnerability Database, Libraries.io, and Google's open source vulnerabilities database. We discovered compelling empirical evidence supporting both the proposed knowledge spillover effect and the moderating relationship. Acknowledging the existence of various causal pathways that may contribute to the observed knowledge spillovers, we analyzed potential mechanisms and showed that our theory (i.e., organizational learning from vulnerability disclosures through software supply chains) was a more plausible and salient mechanism relative to the alternatives.

Keywords: open source security, open source software, software supply chain, software dependency, software vulnerability, information security, security incident, organizational learning, digital sovereignty

"Free and open source software is a vital cog in the economy, much like interstate highways, the power grid, or the communications network. Given how much we already know about those critical infrastructure systems, doesn't it only make sense to learn just as much about their 21st century equivalent?" (Lifshitz-Assaf & Nagle, 2021)

INTRODUCTION

Open source software (OSS) plays a vital role in the current digital economy. Almost all commercial codebases contain open source components (Synopsys, 2022), and most of the core software innovations and frameworks in emerging technologies are OSS projects, such as TensorFlow for artificial intelligence, Ethereum for blockchains, Kubernetes for cloud computing, and Spark for data analytics. Along with their significant presence, recent research has shown that OSS can play salient roles in improving productivity and creating value for commercial firms (Germonprez et al., 2017; Lin & Maruping, 2022; Nagle, 2019).

Despite being widely used and highly valued, OSS is not free from security defects (Altinkemer et al., 2008; Payne, 2002; Schryen, 2011). The availability of source code to the public does not automatically make OSS more secure. Because anyone can freely use OSS and can easily incorporate it into other software applications, security issues in OSS are likely more contagious and exploitable than their proprietary, closed source counterparts (Ransbotham, 2010).[1] The "Log4Shell" vulnerability in the Apache Log4j library discovered in December 2021 was an illuminating example. Log4j is a popular open source logging utility for Java programs. Any software that uses Log4j can be potentially vulnerable and affected by Log4Shell because attackers can execute commands remotely on the target machine to steal data, install malware, or take control. The impact of Log4Shell was extensive because there are billions of

[1] We acknowledge that some OSS is proprietary or commercial, and some proprietary and commercial software is open-sourced. However, for ease of exposition, throughout the paper we consider OSS as noncommercial and nonproprietary, and commercial and proprietary software as closed source software. This is consistent with the OSS literature and the typical software use cases in practice. We use the phrases "closed-source software," "proprietary software," and "commercial software" interchangeably in this paper as an alternative group of software artifacts when compared to OSS.

devices that run on Java (including computers, phones, ATMs, and home appliances) and logging is a universal software activity. In the wake of this discovery, the White House promptly convened government agencies and private sector stakeholders to discuss how to prevent security defects and vulnerabilities in OSS and improve the process of finding defects and fixing them.[2] Meanwhile, newly passed laws and regulations, such as the CHIPS and Science Act (signed into law by President Joe Biden on August 9, 2022) and the Securing Open Source Software Act (introduced by the leadership of the Senate Homeland Security and Governmental Affairs Committee on September 21, 2022), have included provisions to strengthen open source security and software supply chains. Indeed, flaws in OSS can threaten national security. Just as nations strive to safeguard their physical infrastructure, there is a growing consensus on the need to secure the OSS ecosystem because it is key to digital sovereignty (Berlin Declaration, 2020) and underpins much of the digital infrastructure in the modern economy (Eghbal, 2020; Lifshitz-Assaf & Nagle, 2021).

In this study we seek to deepen the understanding of open source security, doing so through the perspective of *software supply chains*. Some researchers have already noticed that OSS operates in complex supply chains (Germonprez et al., 2017). The aforementioned Log4j incident also highlights the fact that OSS packages are intricately interconnected, and as such, open source security concerns more than just fixing defects within the scope of individual OSS projects. In fact, most OSS projects reuse external open source code and libraries written by others to minimize redundant effort. According to a recent study by the Linux Foundation (2022), on average, an OSS project imports and reuses 68.8 external open source packages. The interconnected relationships among OSS projects naturally engender a unique ecosystem with

[2] See https://www.whitehouse.gov/briefing-room/statements-releases/2022/01/13/readout-of-white-house-meeting-on-software-security/

multilateral dependencies (Jacobides et al., 2018). In this ecosystem, it is fitting to view the software dependency relationships between OSS packages as supplier–consumer dyads, in which a *supplier* is defined as a software package that provides functionalities for other downstream software packages, whereas a *consumer* is a software package that utilizes the functionalities provided by upstream software packages.[3] The role and importance of software supply chains are therefore salient characteristics of open source security because security vulnerabilities in an OSS artifact have impacts and implications for other downstream OSS artifacts.

To make progress in understanding how OSS projects manage and respond to security vulnerabilities, this paper advances a theory of open source security that goes beyond the boundaries of individual OSS projects and considers the influences of their upstream suppliers. Viewing OSS from the perspective of software supply chains and drawing on organizational learning theory (Mehrizi et al., 2022), we propose, explain, and investigate the spillover of security knowledge in vulnerability disclosures through software supply chains in the OSS ecosystem: when a supplier discloses a security vulnerability, its downstream consumers will apply the security knowledge in the disclosed vulnerability to identify similar vulnerabilities in the future. This work is related to prior research on the spread of security vulnerabilities after their public disclosures (see, e.g., Mitra & Ransbotham, 2015). However, our research is distinct from, and therefore, complementary to, this literature in two key aspects. First, we emphasize the diffusion of security knowledge from the OSS supplier to its OSS consumers. This allows us to develop a more nuanced understanding of the direction of knowledge diffusion. Second and perhaps more importantly, in our setting the supplier's vulnerability and the consumer's

[3] We are not the first one to use "software supply chains," "suppliers," and "consumers" to characterize open source security. They have been the standard terminology in prior research (Ferraiuolo et al., 2022), recent cybersecurity policies in the U.S. (Executive Office of the President, 2021), as well as industrial initiatives and reports (Synopsys, 2022). We adopt the supply chain terminology in accordance with this ongoing discourse.

vulnerability were not the same. In other words, we examine the spread of knowledge related to security weaknesses in a vulnerability in the open source ecosystem rather than the spread of the vulnerability itself.[4]

We assembled a unique dataset that combined software vulnerability records and OSS project data to test our theory. Results from our empirical analysis were consistent with our theoretical propositions. We found that OSS consumers were significantly more likely to report the same security weaknesses that their direct suppliers had previously disclosed. We further showed that the severity of the vulnerability moderated knowledge spillover. The disclosures of critical vulnerabilities, relative to noncritical ones, yielded a substantially higher degree of interorganizational learning and knowledge spillovers in the OSS ecosystem.

Overall, this research makes two main contributions. First, we develop a novel theoretical framework for understanding open source security. Through the lenses of organizational learning and software supply chains, we explain and show how software dependencies become channels for knowledge spillovers, such that OSS projects learn from their suppliers to improve the security of their own software. Second, this paper helps bridge two enduring research topics—OSS development and information security—in the information systems (IS) scholarship that had been largely disconnected. As open source security becomes an increasingly salient issue in the digital society, IS researchers can offer important insights to this ongoing critical dialogue and compelling implications for practitioners and policymakers about how to secure the open source ecosystem. Our work represents one of the first steps in this direction.

[4] Take the Log4Shell vulnerability for example: the scope of this study concerns the spread of security knowledge related to the technical roots of the vulnerability (e.g., improper input validation, uncontrolled resource consumption, and so on) from Log4j (the supplier) to its downstream consumers OSS packages, rather than the spread of the Log4Shell vulnerability on the internet.

CONCEPTUAL BACKGROUND

Open-Source Development and Software Supply Chains

Over the past two decades, OSS has become a mainstream approach to software development, attracting support and engagement from individual developers, governmental and nongovernmental organizations, and commercial corporations. This open source movement has been coupled with, and reinforced by, the emergence of digitalization. On the one hand, OSS embodies a new organizing logic of digital innovation whereby the innovation processes involve fluid boundaries, and distributed agencies and the innovation outcomes are modular and reprogrammable (Nambisan et al., 2017; Yoo et al., 2010). On the other hand, digital platforms such as GitHub assist in standardizing and streamlining the workflows for developing OSS projects and enacting socio-technical affordances to address the challenges of knowledge exchange, deliberation, and combination in large-scale collaborations (Malhotra et al., 2021).

Prior research has explored many theoretical underpinnings of OSS development. Earlier work in this area sought to understand the intrinsic and extrinsic motivations behind voluntary contributions by software developers (Roberts et al., 2006; von Krogh et al., 2012) and has examined the governance mechanisms and dimensions in such a distributed, community-based environment (O'Mahony & Ferraro, 2007; Tullio & Staples, 2013). Researchers have also provided significant insights into the coordination among distributed actors and tasks within OSS projects through routines and open superposition (Germonprez et al., 2021; Howison & Crowston, 2014; Lindberg et al., 2016). As corporate involvement in OSS projects becomes increasingly common, a number of scholars have begun to examine how firm attributes, such as ideology and credibility, affect the relationship between commercial firms and OSS communities (Daniel et al., 2018; Spaeth et al., 2015), and why OSS engagements create value for the business (Germonprez et al., 2017; Lin & Maruping, 2022; Nagle, 2018).

An important topic that has received less discussion in the prevailing OSS literature and IS research in general is the notion of software supply chain, which was defined as "the network of stakeholders that contribute to the content of a software product or that have the opportunity to modify its content" (Alberts et al., 2011, p. 2). A primary mechanism through which software supply chains are formed is by importing external software artifacts into one's own software.[5] This allows software developers to directly apply and execute the code written by others so that they do not need to reinvent the wheel. This type of code reuse is pervasive in OSS development (Haefliger et al., 2008; Stanko, 2016) and gives rise to complex software supply chains that manifest not only during the development and testing phases of the software but also in the production environment (Cox, 2019). Through this massive network of software dependencies, OSS projects benefit from reusing external software to reduce costs and increase efficiency in OSS development. The downside, however, is that any software vulnerability in an OSS artifact can rapidly propagate across the network through multiple layers of dependencies. This could impact every OSS artifact downstream in the software supply chain.

Although software supply chains are present in the OSS ecosystem as well as proprietary software products (Ellison & Woody, 2010), there are functional and structural differences in the software supply chains of OSS compared to the ones in proprietary software. In terms of the functional differences, proprietary software often requires formal licensing agreements and fees, which come with some level of warranty for the software. To avoid legal and financial liabilities

[5] Another common type of code reuse is forking. By forking, software developers make a copy of the content in an existing OSS repository. This allows anyone to take full control of the codebase so as to modify or experiment with the code. Forking is particularly useful in enabling external developers without a write access to a repository to easily propose changes (e.g., fixing a bug or adding a feature) and contribute back to the source repository through a pull request. From the perspectives of open source security and software supply chains, forking relationships are much less salient compared to dependency relationships because the former are less common in production environments. Moreover, forking relationships are much simpler than dependency relationships for two reasons. First, an OSS project can be forked from one existing project, but an OSS project typically has many software dependencies (The Linux Foundation, 2022). Second, forking relationships rarely go beyond one degree (A → B), whereas dependency relationships typically extend beyond one degree (A → B → C → D). As such, we focus on dependency, rather than forking, relationships in this study.

from their software defects, proprietary software vendors are typically obligated to push software updates and patches to their consumers (Arora et al., 2008; Ellison & Woody, 2010). By contrast, there is no such contractual quality guarantee in the use of OSS, and as such, it is the consumers' responsibility to monitor and update their dependencies when a new version is made available by their suppliers (Synopsys, 2022). As for structural differences, MacCormack et al. (2006) showed that OSS software architecture is more modular than proprietary software architecture because OSS is typically developed by a distributed team, whereas proprietary software tends to involve a collocated team of developers. This difference in software designs can directly affect the number of dependencies and their pattern of distribution (MacCormack et al., 2006), subsequently shaping the structure of their respective software supply chains. Furthermore, proprietary software, unlike OSS, typically discourages, even prohibits, consumers from viewing, modifying, or redistributing the source code. This should make proprietary software less accessible and remixable, which in turn reduces its likelihood of becoming a dependency of another software. As a result, the software supply chains in the OSS ecosystem are expected to be longer (i.e., more layers of dependencies) and broader (i.e., higher number of dependencies in a software artifact) when compared to the ones in the context of proprietary software development.

Such functional and structural characteristics of the software supply chains in the OSS ecosystem have implications for organizational learning and the management of vulnerable dependencies. On the one hand, OSS consumers are expected to take a more active and conscious role in monitoring and applying software updates from their suppliers (Prana et al., 2021). This should better effectuate the processes and outcomes of learning from supplier's vulnerabilities. On the other hand, OSS consumers naturally have more opportunities than their commercial counterparts to acquire knowledge from their suppliers because OSS tends to have

more external dependencies; at the same time, the suppliers' vulnerable code and their corresponding fixes are made available for the consumers to access and review. In summary, software supply chains in the OSS ecosystem are expected to play a greater role in knowledge transfer and learning in OSS projects than in proprietary software development.

Software Vulnerabilities

Software vulnerabilities are defects in software code that can be exploited by an attacker to make the software act in unintended and unexpected ways. The prevalence and economic impact of software vulnerabilities make them an enduring topic in information security research.[6] Much of this research has been focused on four interrelated aspects of software vulnerability management: *discovery*, *disclosure*, *diffusion*, and *patching*. Discovering the existence of a vulnerability is typically the first step in software vulnerability management (Ransbotham et al., 2012). Prior research shows that characteristics of the discoverer, the vulnerability, and the software can affect disclosure timing (Sen et al., 2020) as well as patching behavior (Arora et al., 2010). Because the vast majority of software vulnerabilities were discovered by non-malicious actors, it is now a standard cybersecurity practice to withhold public disclosure until a patch for the vulnerability is available (Sen et al., 2020). Results from prior analytical and empirical research suggest that publicly disclosing a software vulnerability can accelerate patch release, on the one hand, and the diffusion of attacks seeking to exploit the vulnerability, on the other hand (Arora et al., 2010; Mitra & Ransbotham, 2015). Although patch releases are critical in addressing vulnerabilities, Arora and Telang (2005) found that they are also associated with a spike in attacks, suggesting that attackers are targeting users of the software who did not promptly patch

[6] The 2017 Equifax data breach was a high-profile example showing the economic impact of software vulnerabilities. It was caused by an unpatched software flaw in Apache Struts, an open source framework for developing web applications. The flaw allowed remote attackers to execute arbitrary commands and, as a result, compromised sensitive personal information of nearly 150 million Americans, according to Equifax.

the vulnerability.

The existing knowledge regarding software vulnerabilities in OSS is relatively limited. Prior research on open-source security has primarily focused on two categories of questions: whether OSS is more secure than closed-source software, and how OSS developers react to software vulnerabilities differently compared to their closed-source counterparts. In general, researchers found no substantial differences between OSS and closed-source software in terms of the risk and severity of software vulnerabilities (Altinkemer et al., 2008; Payne, 2002; Schryen, 2011), but there was some evidence that the OSS may release patches faster than closed-source software (Arora et al., 2010; Temizkan et al., 2012). As such, Schryen (2011, p. 139) suggested that "we should explore other factors rather than asking whether open source or closed source software leads to higher levels of security."

The unique nature of OSS challenges some of the assumptions in prior analytical models for optimal disclosure and patching policies. For example, OSS licenses typically explicitly state that the licensor provides the work on an "as is" basis, without warranties of any kind. As such, OSS projects do not financially internalize any customer losses, making it difficult to derive the optimal disclosure policy from customer losses as proposed by Arora et al. (2008). Similarly, with code contributed by volunteer developers, it is not straightforward to use the patch development costs (cf, Cavusoglu et al., 2007) to study patch release policies in OSS because the costs could be virtually zero in OSS settings. Thus, research on software vulnerabilities in OSS would require a new conceptual framework beyond the conventional view centered on commercial software vendors.

Meanwhile, the open source ecosystem has its own unique challenges and considerations as regards software vulnerabilities. For example, many OSS developers do not prioritize dependency updates when a new version of the dependencies becomes available (Kula et al.,

2018). This is because dependency updates may induce breaking changes. Therefore, this requires extra migration effort for OSS developers to test and fix any incompatibility issues that come with the new software dependencies. With inadequate resources and unpaid labor, OSS developers are often less attentive to dependency management tasks (Bogart et al., 2015; Eghbal, 2020). By the same token, research has shown that OSS project maintainers were less likely to invest in preemptive security practices, such as security training and audits, and instead relied heavily on community support to address security issues (Pashchenko et al., 2020).

To summarize, software vulnerabilities in OSS are important, but understudied. Despite the rich analytical and empirical insights in the software vulnerability literature, it remains unclear how the security of OSS should be conceptualized and improved. A potentially fruitful theoretical lens for studying software vulnerabilities in OSS is organizational learning from security incidents, which we will elaborate on as follows.

Organizational Learning from Security Incidents

Software vulnerabilities and other security incidents often provide opportunities for organizational learning. In their recent review of how organizations learn from IS incidents, Mehrizi et al. (2022) developed an integrated conceptual framework with three distinct learning modes: *reflective* (i.e., learning from the past for the future), *embedded* (i.e., learning from the present for the present), and *prospective* (i.e., learning from the future for the future). They observed that this literature has predominantly been concentrated on reflective learning from incidents, emphasizing post-incident analysis and incident knowledge dissemination as two important learning practices (see, e.g., Gal-Or & Ghose, 2005; McLaughlin & Gogan, 2018). These two learning practices are distinct from, yet complementary to each other: post-incident analysis seeks to identify root causes and extract generalizable lessons from the incident, whereas incident knowledge dissemination aims to distribute and share incident knowledge

across organizational units and boundaries (Mehrizi et al., 2022). In other words, reflective learning involves a deep reflection of the incident and can take place at the organizational and interorganizational levels (Majchrzak & Jarvenpaa, 2010; Skopik et al., 2016).

Despite this extensive literature on organizational learning from security incidents, prior studies were typically contextualized in canonical and well-bounded organizations, such as commercial firms and governmental agencies. More likely than not, the processes and outcomes of knowledge creation, retention, and sharing in these organizations will differ from those in online OSS communities (Faraj et al., 2011; Safadi et al., 2021). For example, whereas commercial firms often have a dedicated team or department for cybersecurity management and incident response (Ahmad et al., 2020), most OSS projects do not have the resources and capacity to include such dedicated security personnel in the projects. Similarly, although researchers have recognized that learning from security incidents can arise at the interorganizational level (Mehrizi et al., 2022), there is a paucity of theorizing and evidence on how such learning may arise in online OSS communities.

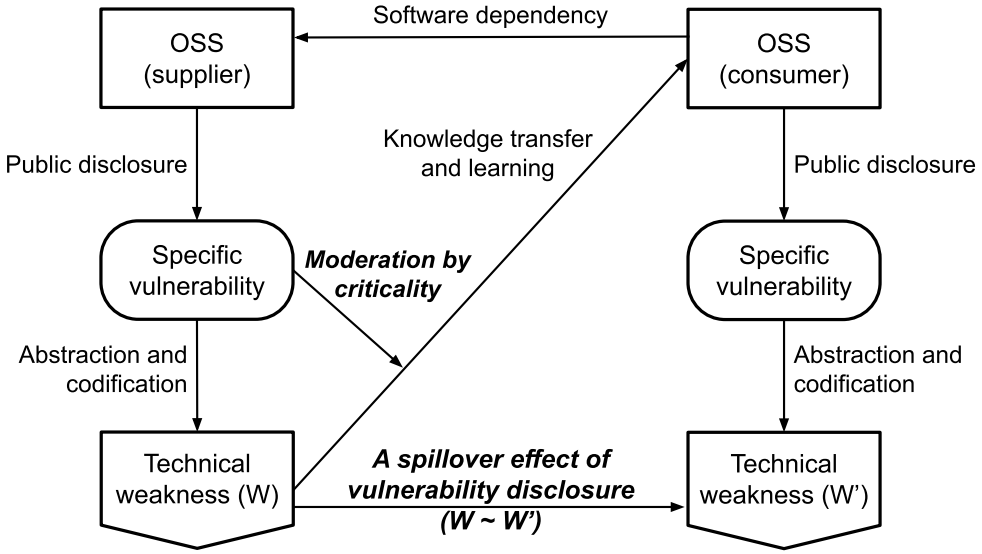
In summary, significant gaps remain in our understanding of organizational learning from security incidents in the OSS context. Our research contributes to this discourse by theorizing one potential mechanism by which organizational learning and knowledge spillover may manifest from vulnerability disclosures through software dependencies in the open source ecosystem.

THEORY DEVELOPMENT

In this section, we develop a framework for understanding knowledge spillovers from vulnerability disclosures in the open source ecosystem, as depicted in Figure 1. We will argue that through the mechanism of organizational learning, OSS projects are more likely to identify and disclose security weaknesses that had been revealed earlier in their direct suppliers'

vulnerability disclosures. Specifically, we will explain why the knowledge transfer and learning would occur across the software supply chain from a supplier to its consumers and how the criticality of the disclosed software vulnerability from the supplier may moderate such learning at the interorganizational level.

Figure 1. Conceptual Model



Knowledge Spillovers from Vulnerability Disclosures

When an OSS project discloses a vulnerability, the purpose is first and foremost to increase public awareness of the specific vulnerability in the software so that its users, including downstream projects in the software supply chain, can take timely remediation actions, such as patching or removing the affected software asset, to avoid potential exploits. Beyond this primary purpose of addressing the incident at hand, vulnerability disclosures can provide opportunities for learning in other OSS projects. We propose a spillover effect of vulnerability disclosures: OSS projects are more likely to identify and disclose security weaknesses that were previously disclosed by their direct OSS suppliers. At the center of our reasoning are two interrelated mechanisms in vulnerability disclosures and the software supply chains: (1) the abstraction and codification of technical weaknesses in vulnerability disclosures and (2) the

transfer of security knowledge through software dependencies. As we explain in the following, they are consistent with post-incident analysis and incident knowledge dissemination in reflective learning from security incidents (Mehrizi et al., 2022).

Vulnerability disclosures from OSS projects as well as other proprietary software vendors typically involve some degree of knowledge abstraction and codification. Theory and evidence in the organizational learning literature suggest that abstraction and codification can make knowledge more readily transferable (Argote, 2013; Zander & Kogut, 1995; Zollo & Winter, 2002). Besides incident-specific and response-oriented information (for example, how attackers can exploit the vulnerability, what versions of the software are affected, and where to obtain a patch), it has been a norm in the vulnerability disclosure process for software owners and security analysts to reflect on the specific vulnerability and use the common weakness enumeration (CWE) coding scheme to characterize the technical core of the security vulnerability (Schryen, 2011).[7] As a community-developed coding scheme, CWE is intended to facilitate communication among diverse stakeholders about vulnerabilities and exposures in computer software. To this point, Boh (2007) has suggested that institutionalized-codification mechanisms (such as template, database, and standardized methodology) are most suitable for knowledge sharing among geographically dispersed project-based organizations when the nature of work is standardized. This is also consistent with Nonaka's theory of organizational knowledge creation that "explicit" knowledge is actionable across contexts and accessible through consciousness (Nonaka & von Krogh, 2009). These codified technical weaknesses in OSS suppliers' vulnerability disclosures should help develop abstract, generalizable lessons and identify systematic causes, as in conventional post-incident analysis.

[7] See https://cwe.mitre.org/about/index.html

Although codified security knowledge is more easily transferable, the direction of knowledge transfer will likely be affected by contextual factors within and across organizations (Argote, 2013). Interorganizational learning and knowledge spillover from a focal organization to other organizations in the environment have been well documented in prior research in non-OSS settings, such as commercial corporations and academic research centers (Argote et al., 2021; Autio et al., 2004; Zollo & Reuer, 2010). Consistent with this literature, we argue that OSS projects are more likely to acquire knowledge from their direct suppliers. It is important to note that in the United States alone, there are more than 10,000 vulnerabilities being discovered and disclosed every year in the National Vulnerability Database. Although these vulnerability disclosures are released to the public and made available to everyone, it is unlikely that OSS projects keep track of all these disclosures, especially given their lack of resources and labor (Geiger et al., 2021). As such, OSS projects would likely pay more attention to vulnerability disclosures from a narrow set of selected sources. When deciding which sources to prioritize, OSS projects will most likely prioritize those with which they have a direct relationship. One apparent choice is the OSS projects they depend on—that is, their direct OSS suppliers. Hansen (1999) has found that a strong tie between the two parties is often required for transferring complex knowledge. As a result, OSS projects would be more likely to pay attention to their suppliers' vulnerability disclosures and subsequently acquire security-related knowledge from there. This will increase the likelihood that these consumers will identify and disclose similar security weaknesses that were previously disclosed by their suppliers.

In summary, with knowledge abstraction and codification in vulnerability disclosures and software dependencies as channels for knowledge transfer in the OSS ecosystem, we propose:

> **PROPOSITION 1.** *OSS consumers are more likely to disclose security weaknesses that were previously disclosed by their direct OSS suppliers.*

Moderation by Criticality

Software vulnerabilities have varying degrees of severity, ranging from low to critical. The information security literature has shown that the severity of a software vulnerability has significant implications, affecting the timing of vulnerability disclosure (Sen et al., 2020), the propagation of the vulnerability (Chinthanet et al., 2021), the timeliness of patch release (Arora et al., 2010), and the efficacy of organizational learning (Ahmed & Lee, 2020).

OSS projects will likely perceive a more profound need for learning when their suppliers disclose critical software vulnerabilities and better retain the "lessons learned" from these critical software vulnerabilities (Dahlin et al., 2018; Madsen & Desai, 2010). This is consistent with the evidence from the organizational learning literature that organizations are more attentive to learning from errors with relatively severe consequences (Homsma et al., 2009). On this point, Frese and Keith (2015) provided a clear and compelling justification. In their words:

> *Errors with strong negative consequences attract attention and indicate unequivocally that something needs to [be] done; this then leads to learning because people reflect on and discuss their errors, changing routines and understanding. Successful actions or errors with small negative consequences do not indicate a necessity for change. Therefore, errors with small consequences or those that can be corrected immediately are more easily overlooked or ignored.* (p. 676)

In addition, critical vulnerabilities are more likely to induce external and internal communications. Externally, the degrees of press coverage and social media discussions should be higher regarding critical software vulnerabilities in an OSS project (Drupsteen & Guldenmund, 2014). As such, they are less likely to be omitted or ignored by the downstream OSS consumer projects. Internally, critical vulnerabilities, especially from their suppliers, are more likely to promote discussion and reflection among the maintainers of an OSS project. This should better enable knowledge retention and its subsequent use in identifying similar vulnerabilities within their own projects (Argote, 2013).

In view of the aforementioned, it can be argued that critical software vulnerabilities have the potential to intensify knowledge transfer, retention, and use. This will moderate the knowledge spillovers that we proposed earlier.

> **PROPOSITION 2.** *The degree of knowledge spillovers as proposed in Proposition 1 will be more pronounced when the disclosed vulnerabilities from the OSS suppliers are critical*.

EMPIRICAL STRATEGY

Data

To test our propositions, we assembled a unique dataset by integrating data from three sources: the National Vulnerability Database (NVD), Libraries.io, and Google's open source vulnerabilities database.

The NVD is the de facto vulnerability disclosure system. Hosted by the U.S. National Institute of Standards and Technology, it has been the most complete source of public vulnerability disclosures and used extensively in prior software vulnerability research (e.g., Arora et al., 2010; Mitra & Ransbotham, 2015; Ransbotham et al., 2012). Each vulnerability record in NVD contains a wide variety of information. Table 1 uses the Log4Shell vulnerability as an example to illustrate some of the key elements available in a vulnerability record in NVD. Specifically, each vulnerability disclosure in NVD is uniquely identified by a common vulnerabilities and exposures (CVE) number and comes with a publication date to indicate when a vulnerability was disclosed to the public. A vulnerability report also often contains a list of weaknesses based on the CWE coding scheme to characterize the technical roots of the vulnerability (Schryen, 2011). The severity base score in a vulnerability report is a numerical (0–10) representation of the severity and risk of the focal security vulnerability assigned by NVD analysts, following the Common Vulnerability Scoring System (CVSS) industry standard. In practice, a vulnerability is considered critical when its severity base score is 9 or greater.

Table 1. Key Elements in the Log4Shell Vulnerability Report in NVD

CVE Number	CVE-2021-44228
Published Date	12/10/2021
Description	Apache Log4j2 2.0-beta9 through 2.15.0 (excluding security releases 2.12.2, 2.12.3, and 2.3.1) JNDI [Java Naming and Directory Interface] features used in configuration, log messages, and parameters do not protect against attacker-controlled LDAP [Lightweight Directory Access Protocol] and other JNDI-related endpoints. An attacker who can control log messages or log message parameters can execute arbitrary code loaded from LDAP servers when message lookup substitution is enabled. …
Weakness Enumeration	CWE-20: Improper Input Validation CWE-400: Uncontrolled Resource Consumption CWE-502: Deserialization of Untrusted Data …
Severity	Base score: 10.0

The Libraries.io data provide a cross-sectional snapshot of public OSS repositories as of January 12, 2020 (Katz, 2020).[8] Recently, this dataset was used in the Census II report from the Linux Foundation and the Laboratory for Innovation Science at Harvard to understand the usage of OSS packages (Nagle et al., 2022). The Libraries.io data contain rich information about repository-level attributes and statistics, such as programming language and number of contributors. The most unique aspect of the Libraries.io data is that they aggregate dependency relationships among OSS projects from many different package managers, yielding over 235 million interdependencies among 33 million repositories. This enabled us to identify supplier–consumer dyads in the OSS ecosystem. In our study, the OSS suppliers and consumers were identified at the level of individual repositories. This avoids the issue that an OSS project may involve multiple repositories and enables us to better capture software dependencies and incorporate repository-level controls for each project.

To achieve a mapping between the vulnerability records from NVD and the OSS data from Libraries.io, we relied on Google's open source vulnerabilities database. This database has been released and maintained by Google since 2021.[9] It provides a crosswalk between CVE

[8] https://libraries.io/data

[9] See https://opensource.googleblog.com/2021/02/launching-osv-better-vulnerability.html. It is important to note that this

numbers and OSS package names.

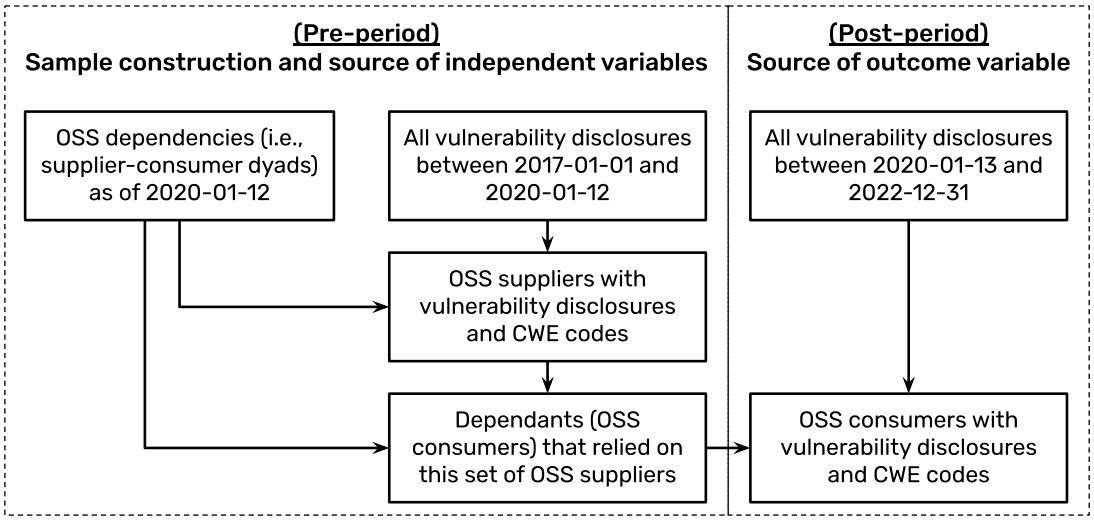
Research Design and Sample Construction

We distinguished two time periods in our research design, using January 12, 2020, when the Libraries.io data were released, as the cutoff date (Figure 2). We used data from the pre-period (between 2017-01-01 and 2020-01-12) to construct our sample, detect supplier–consumer dependencies, and identify disclosed vulnerabilities and weaknesses in OSS suppliers. The OSS projects and dependencies were version agnostic in our research design (e.g., "Log4j" rather than "Log4j-2.15.0"). This was to circumvent the complexities associated with analyzing software versions (Nagle et al., 2022) and make the analysis computationally feasible.[10] The OSS suppliers we considered were OSS artifacts that had a vulnerability disclosure in the pre-period, and the OSS consumers were OSS artifacts that directly depended on these suppliers in the pre-period. In other words, our sample was from supplier–consumer dyads that had a direct dependency, and we omit indirect software dependencies in our study. From this pool of OSS consumers, we used data from the post-period (between 2020-01-13 and 2022-12-31) to identify their vulnerability disclosures and technical weaknesses.

database is different from the Open Source Vulnerability Database used in prior research (e.g., Sen et al., 2020). Though both databases serve similar purposes, the latter was launched in 2004 by Jake Kouhns and shut down in 2016.

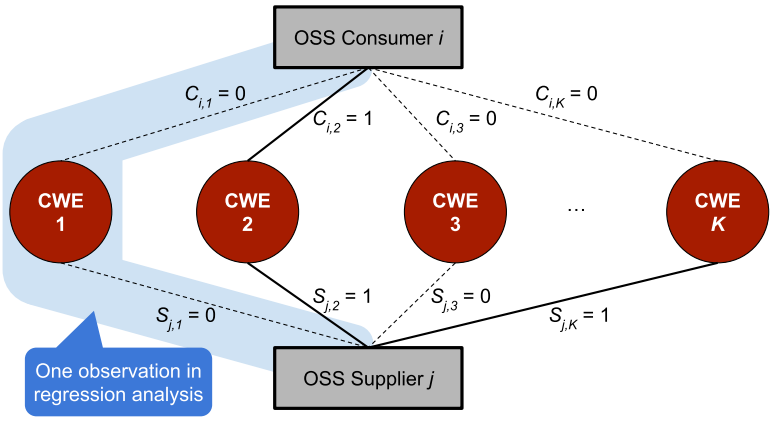
[10] We found that a versioned dependency network of our sample OSS projects was about 25 times larger than an unversioned one. Given that our analysis on unversioned supplier–consumer dyads required over 40GB of computer memory, the analysis of versioned supplier–consumer dyads likely would require at least 1TB of computer memory, assuming that the amount of computer memory needed grows linearly with the size of the input data.

Figure 2. Research Design



Given that a CVE report can have multiple CWE codes, the unit of observations in our regression analysis was at the level of supplier-CWE-consumer triads. This allowed us to encode the presence (or absence) of a specific security weakness in an OSS consumer in the post-period and the presence (or absence) of the weakness in its OSS supplier in the per-period. Figure 3 provides a schematic illustration of our data structure in regression analysis, which, as we elaborate next, allows us to capture CWE-specific knowledge transfer between supplier–consumer dyads and control the heterogeneity in CWE codes.

Figure 3. Illustrating the Unit of Observations at the Suppler-CWE-Consumer Level in Regression Analysis



$C_{i,k}$: a dummy indicating whether consumer i reports CWE code k in the post-period
$S_{j,k}$: a dummy indicating whether supplier j reported CWE code k in the pre-period

Methodology

With our research design, our analytical objective was to examine whether the technical weaknesses (i.e., CWE codes) in the consumers' vulnerability disclosures in the post-period were the same as the ones in the vulnerability disclosures from their direct suppliers in the pre-period. Accordingly, we estimated the following regression model:

$$C_{i,k} = \beta S_{j,k} + \gamma \boldsymbol{X}_i + \delta \boldsymbol{W}_j + \alpha_k + \mu_i + \varepsilon_{i,k},$$

where *i*, *j*, and *k* denote consumer, supplier, and weakness, respectively. The outcome variable, $C_{i,k}$ (*CONSUMER DISCLOSES CWE k*), was a dummy indicating whether OSS consumer *i* reports a type *k* weakness in the *post*-period. The main independent variable was $S_{j,k}$ (*SUPPLIER DISCLOSED CWE k*), which was a dummy indicating whether the supplier *j* reported a type *k* weakness in the *pre*-period. Our primary interest was in the coefficient β, which detected whether $S_{j,k}$ was associated with $C_{i,k}$. The $\boldsymbol{X}_i$ and $\boldsymbol{W}_j$ were vectors of log-transformed baseline control variables as of 2020-01-12 from consumer *i* and supplier *i*, respectively. These included *AGE* (difference, in years, between repository creation date and 2020-01-12), *STARS COUNT* (number of stars on the repository), *CONTRIBUTORS COUNT* (number of unique contributors that have committed to the repository), and *DEPENDENTS COUNT* (number of other projects that declared the project as a dependency). These control variables reflected the likelihood of security vulnerability and the health of the focal OSS community (Arora et al., 2010; Geiger et al., 2021; Goggins et al., 2021). We also included a dummy, *CRITICAL VULN*, in $\boldsymbol{W}_j$ to indicate whether the disclosed vulnerability was critical (i.e., CVSS base score of 9 or above) for the OSS suppliers. Finally, we incorporated two sets of fixed effects (FEs) in our empirical model. First, we used CWE FEs, α_k, to account for time-invariant, unobservable characteristics of the weaknesses. Since some weaknesses may be easier

to discover or more common than others in OSS development, the CWE FEs would capture and address these unobservable weakness-specific factors in our estimation. We also included FEs for the consumer's programming language, μ_i.[11] Different programming languages tend to have distinct security properties (e.g., Go is a memory-safe language while C is not). At the same time, the norms and standards in software development and maintenance practices vary depending on the programming languages (Decan et al., 2019). Such language-specific unobservables would be controlled and absorbed by μ_i.

Aside from the main model, our Proposition 2 involves a moderation effect. We tested it by adding an interaction term between $S_{j,k}$ and *CRITICAL VULN* into the main model. A significant coefficient for the interaction term would suggest that the criticality of the supplier's vulnerability can moderate the spillover of security knowledge over software supply chains.

For identification purposes, we further imposed a few restrictions when constructing our research data. First, we focused on OSS projects hosted on GitHub to obtain relevant control variables and avoid the potential discrepancy in OSS vulnerability disclosures caused by the project hosting sites. Second, we limited OSS consumers to those that had not disclosed any vulnerabilities in the pre-period to avoid potential impacts from such earlier disclosures in the post-period. Third, we excluded supplier–consumer dyads when the two projects were created by the same organization, in accordance with the scope of our theorizing of knowledge transfer and learning at the interorganizational level.[12] Fourth and finally, because we identified security weaknesses through CWE codes in vulnerability disclosures and because the vulnerability

[11] We did not include FEs for the supplier's programming language because in most cases, supplier and consumer are using the same programming language. The results of our estimation were greatly similar if we included FEs for the supplier's, instead of the consumer's, programming language in our model.

[12] For example, "rails/rails" had a dependency on "rails/sprockets," and as such, the latter was considered as a supplier of the former. However, we excluded this supplier–consumer dyad because both OSS projects were developed within the same organization "rails."

disclosures from the suppliers and the consumers are from two different time periods, we limited our analysis to CWE codes that appear in both pre- and post-periods. This removed new (only in the post-period) and potentially outdated (only in the pre-period) CWE codes and enabled us to incorporate the CWE FEs into our empirical model.

In all, our final sample comprised 385,722 supplier–consumer dyads, with 430 unique suppliers and 288,132 unique consumers. The suppliers collectively disclosed 1,292 vulnerabilities in the pre-period, and the consumers disclosed 865 vulnerabilities in the post-period. These numbers may seem small, but this was because we purposefully removed certain suppliers, consumers, supplier–consumer dyads, and vulnerabilities in our empirical analysis, as we described earlier, to be able to better identify the proposed organizational learning effects. Also, as we will further elaborate and verify later in our analysis of mechanisms, it is important to note that these suppliers and consumers did not have the exact same vulnerability because the vulnerabilities of the suppliers and consumers came from different CVE reports.[13] Table 2 provides a summary of the data statistics for our variables.

[13] To see this, consider the Log4Shell vulnerability. Even though Log4j is widely used and Log4Shell impacted virtually all Log4j's consumers, none of these consumers will have a CVE report on the Log4Shell vulnerability because the CVE assignment rules prohibit duplicated CVE reports for the same vulnerability. Since we identified CWEs from suppliers' and consumers' CVE reports and the suppliers and consumers will not have CVE reports for the exact same vulnerability, our design is robust against the concern that the CWE codes shared by the supplier–consumer dyads may due to the diffusion of software vulnerabilities.

Table 2. Data Statistics

Variable	Dummy	Obs.	Mean	SD	Min	Max
CONSUMER DISCLOSES CWE k (i.e., $C_{i,k}$)	Yes	29,644,056	0.00	0.01	0	1
SUPPLIER DISCLOSED CWE k (i.e., $S_{j,k}$)	Yes	29,644,056	0.03	0.16	0	1
CONSUMER AGE	No	29,644,056	3.23	2.14	0	12
CONSUMER STARS COUNT	No	29,644,056	133.25	1514.87	0	155566
CONSUMER CONTRIBUTORS COUNT	No	29,644,056	6.10	30.57	0	2647
CONSUMER DEPENDENTS COUNT	No	29,644,056	31.25	1555.12	0	451563
SUPPLIER AGE	No	29,644,056	8.27	2.03	1	12
SUPPLIER STARS COUNT	No	29,644,056	25359.58	22889.06	0	137880
SUPPLIER CONTRIBUTORS COUNT	No	29,644,056	316.35	186.04	0	1247
SUPPLIER DEPENDENTS COUNT	No	29,644,056	54866.82	53343.54	0	151954
CRITICAL VULN	Yes	29,644,056	0.29	0.45	0	1

Econometrically, our data were cross-sectional. Estimating causal relationships from cross-sectional data is typically difficult because of the issue of reverse causality and the impact of unobservables. However, it is important to note that the "treatment" variable, $S_{j,k}$, and the "outcome" variable, $C_{i,k}$, in our design came from two different time periods. This should alleviate the concern about reverse causality. Furthermore, $S_{j,k}$ was likely exogenous in our regression because, for the most part, OSS consumers have no control over whether, when, or what security vulnerabilities their OSS suppliers disclose. One may have the concern that OSS consumers could affect their OSS suppliers' vulnerability disclosures by reporting security bugs to their suppliers in the pre-period. As we will show, there is no substantial evidence that the OSS consumers in our sample contributed to their suppliers in the pre-period. More importantly, if they did, this in theory would attenuate our estimated β coefficient because, having been attentive to such security weaknesses in the pre-period, these OSS consumers would be less likely to develop code with these technical weaknesses in the first place, making it more difficult for us to detect an effect of $S_{j,k}$ and yielding a more conservative estimate for the β coefficient.

RESULTS

Main Results

Table 3 reports the results of our estimation. From columns 1 and 2, we find that the coefficients of $S_{j,k}$ are positive and significant in models without and with control variables. Therefore, Proposition 1 is supported, indicating that OSS projects are more likely to identify and disclose security weaknesses that were previously disclosed by their direct OSS suppliers. The results from column 3 show a positive and significant coefficient for $S_{j,k}$ * *CRITICAL VULN*. This supports Proposition 2 and suggests that the criticality of suppliers' vulnerability moderates the knowledge spillover. Interestingly, the coefficient of the interaction term (0.451) is nearly three-quarters that of the coefficient of $S_{j,k}$ (0.605). This means that in both statistical and practical aspects, the disclosure of critical vulnerabilities by OSS suppliers induces a significantly higher degree of knowledge spillovers to consumers when compared to the disclosure of noncritical vulnerabilities.

Table 3. Main Results

DV: $C_{i,k}$ (*CONSUMER DISCLOSES CWE k*)	(1) Main effect only	(2) Full model	(3) Moderation
$S_{j,k}$ (*SUPPLIER DISCLOSED CWE k*)	1.522***	0.702**	0.605*
	(0.340)	(0.261)	(0.280)
Log *CONSUMER AGE*		-0.247	-0.246
		(0.274)	(0.274)
Log *CONSUMER STARS COUNT*		0.481***	0.481***
		(0.046)	(0.046)
Log *CONSUMER CONTRIBUTORS COUNT*		0.655***	0.654***
		(0.096)	(0.095)
Log *CONSUMER DEPENDENTS COUNT*		-0.084**	-0.083**
		(0.032)	(0.031)
Log *SUPPLIER AGE*		0.208	0.217
		(0.176)	(0.177)
Log *SUPPLIER STARS COUNT*		0.105	0.106
		(0.068)	(0.068)
Log *SUPPLIER CONTRIBUTORS COUNT*		-0.231**	-0.232**
		(0.073)	(0.074)
Log *SUPPLIER DEPENDENTS COUNT*		-0.147***	-0.146***
		(0.034)	(0.033)
CRITICAL VULN		-0.082	-0.187*
		(0.083)	(0.099)
$S_{j,k}$ × *CRITICAL VULN*			0.451***
			(0.098)
CWE FEs	Yes	Yes	Yes
Consumer language FEs	Yes	Yes	Yes
Pseudo R-squared	0.104	0.332	0.333
Number of unique CWEs	78	78	78
Observations	29644056	29644056	29644056

Notes: Robust standard errors (clustered at FE variables) are shown in parentheses. *** p < 0.01; ** p < 0.05; * p < 0.1

Alternative Specifications and Robustness Checks

We implemented two different approaches to assess the robustness of these main results. First, we considered an alternative model specification that included three-way FEs for consumer OSS, supplier OSS, and CWE codes. We did not use this in our main analysis because the three-way FEs would significantly reduce the number of observations in our cross-sectional observations.

More importantly, with this specification, we would not be able to estimate the moderation effect because the moderator, along with all other control variables, would be absorbed by the OSS FEs. Nevertheless, this specification could be useful to address OSS-level unobservables in our estimation. Column 1 of Table 4 shows the results from this three-way FE model. The coefficient of $S_{j,k}$ remains positive (0.755, $p < 0.05$) and has a similar magnitude to the one in column 2 of Table 3 (0.702) without the OSS-level FEs in the estimation. This suggests that the control variables in our main analysis were adequate, and OSS-level unobservables may not be a primary concern in our analysis.

Second, we applied the coarsened exact matching (CEM) procedure to refine the regression sample in our main analysis (Iacus et al., 2011, 2012). Specifically, we matched $S_{j,k}$ based on the control variables. Our CEM procedure used Sturges' rule for automatic bin construction, coarsened the values in each of these control variables into different strata, and retained only strata that had both treated ($S_{j,k} = 1$) and control ($S_{j,k} = 0$) observations. Using the matched sample from this CEM procedure, we reestimated our empirical models and reported the results in columns 2 and 3 of Table 4. Overall, we found that the results with CEM were qualitatively similar to the main results reported earlier.

Table 4. Robustness Checks

DV: $C_{i,k}$ (*CONSUMER DISCLOSES CWE k*)	(1) Three-way FEs	(2) CEM full model	(3) CEM moderation
$S_{j,k}$ (*SUPPLIER DISCLOSED CWE k*)	0.755**	0.624***	0.569***
	(0.325)	(0.192)	(0.212)
Log *CONSUMER AGE*		-0.262	-0.262
		(0.401)	(0.401)
Log *CONSUMER STARS COUNT*		0.540***	0.540***
		(0.080)	(0.080)
Log *CONSUMER CONTRIBUTORS COUNT*		0.588***	0.587***
		(0.152)	(0.152)
Log *CONSUMER DEPENDENTS COUNT*		-0.128***	-0.128***
		(0.036)	(0.036)
Log *SUPPLIER AGE*		0.253**	0.254**
		(0.125)	(0.125)
Log *SUPPLIER STARS COUNT*		0.026	0.026
		(0.071)	(0.071)
Log *SUPPLIER CONTRIBUTORS COUNT*		-0.104	-0.104
		(0.078)	(0.078)
Log *SUPPLIER DEPENDENTS COUNT*		-0.149***	-0.149***
		(0.029)	(0.029)
CRITICAL VULN		0.101	0.068
		(0.108)	(0.121)
$S_{j,k}$ × *CRITICAL VULN*			0.254***
			(0.075)
Consumer FEs	Yes	No	No
Supplier FEs	Yes	No	No
CWE FEs	Yes	Yes	Yes
Consumer language FEs	Yes	Yes	Yes
Coarsened exact matching	No	Yes	Yes
Pseudo R-squared	0.148	0.389	0.389
Number of unique CWEs	78	76	76
Observations	77922	28840176	28840176

Notes: Robust standard errors (clustered at FE variables) are shown in parentheses. In column 1, coefficients for control variables and the interaction term are not available because consumer- and supplier-level FEs absorb their variations. In columns 2 and 3, the number of unique CWEs and observations are different from our main results because the set of valid CWEs (that is, appeared in the pre- and post-periods) changed as a result of CEM. *** p < 0.01; ** p < 0.05; * p < 0.1

Analysis of Mechanisms and Alternative Explanations

To complement our main results, we conducted a set of additional analyses to ascertain that our

proposed theoretical mechanism—that is, organizational learning over software supply chains from OSS suppliers' vulnerability disclosures—is plausible in explaining the observed knowledge spillovers. Table 5 summarizes these additional analyses and their principal results, which we discuss as follows.

Table 5. Analysis of Mechanisms

Alternative explanations/mechanisms	Executed tests	Results
1. OSS projects may be learning from nonsuppliers, and as such, the observed main and moderation effects would exist even without software dependencies.	Implement placebo tests and label CWEs based on alternative vulnerability disclosures published around the same time rather than the ones from the suppliers/consumers.	The placebo results are insignificant, suggesting that software dependencies play a key role in the learning process.
2. The suppliers and consumers could be in the same application categories so that the consumers may learn from others in the same category rather than from their suppliers.	Identify project topics and reestimate the empirical models using only supplier–consumer dyads that had no overlapping topics.	Results show similar patterns and magnitude of coefficients as the main results reported earlier.
3. The vulnerabilities in the consumer OSS projects may be the same as their suppliers' vulnerabilities—that is, the observed main and moderation effects were the diffusion of the suppliers' original vulnerabilities, rather than the results of learning.	Verify whether consumers' vulnerability disclosures mentioned the supplier's CVE number.	We did not find any such instances in our sample.
4. It could be third-party security researchers or random volunteer developers, rather than the core members of the consumer OSS project, who learned from the supplier's vulnerability disclosure.	Conduct qualitative review and coding for the vulnerability discoverers in 30 randomly selected consumer OSS projects.	Most of the discoverers were core members in the consumer OSS projects, suggesting that learning took place within consumer OSS projects.
5. The core members in the consumer OSS projects may contribute to their suppliers' projects and learn from this experience, rather than through suppliers' vulnerability disclosures.	Extend the preceding qualitative analysis to examine whether and how the core members in the consumer OSS projects engaged in their suppliers' projects.	Very few such cases, suggesting that learning by contributing was unlikely the main driver of the knowledge spillovers.

Role of Software Dependencies

Our theorizing hinges on software dependencies as channels for knowledge transfer. As such, it is of critical importance to verify if that is indeed the case. A reasonable and plausible alternative explanation is that OSS consumers may be learning from nonsuppliers, and as such, software dependencies may not be relevant. To address this concern, we examined the role of software dependencies through placebo tests. The rationale of our placebo tests is simple: if the suppliers'

vulnerability disclosures indeed triggered knowledge spillover and organizational learning on the consumers' side, we should *not* detect a significant coefficient for $S_{j,k}$ if the $S_{j,k}$ ($C_{i,k}$) variable does not reflect the security weaknesses of the supplier (consumer).

Following this logic, we implemented two placebo tests, one with a placebo $S_{j,k}$ and the other with a placebo $C_{i,k}$. To derive these placebo quantities, for each vulnerability disclosure from the supplier and consumer we randomly selected another vulnerability disclosure that was published around the same time and used the CWE codes in this "false" vulnerability disclosure to label $S_{j,k}$ in the first placebo test and $C_{i,k}$ in the second one. Table 6 reports the results from these two placebo tests. We find that the coefficients of $S_{j,k}$ and the $S_{j,k}$ × *CRITICAL VULN* interaction term are consistently insignificant. This provides evidence that the weaknesses identified and disclosed by OSS consumers are driven by the weaknesses identified and disclosed by their OSS suppliers. In other words, software dependencies indeed play an essential role in the knowledge spillover process.

Table 6. Placebo Tests

	Placebo supplier CWEs		Placebo consumer CWEs	
DV: $C_{i,k}$ (*CONSUMER DISCLOSES CWE k*)	(1) Full model	(2) Moderation	(3) Full model	(4) Moderation
$S_{j,k}$ (*SUPPLIER DISCLOSED CWE k*)	0.119	0.146	0.079	0.015
	(0.267)	(0.278)	(0.224)	(0.210)
Log *CONSUMER AGE*	-0.214	-0.214	-0.129	-0.129
	(0.251)	(0.251)	(0.268)	(0.267)
Log *CONSUMER STARS COUNT*	0.474***	0.474***	0.488***	0.488***
	(0.048)	(0.048)	(0.040)	(0.040)
Log *CONSUMER CONTRIBUTORS COUNT*	0.674***	0.674***	0.655***	0.655***
	(0.099)	(0.099)	(0.117)	(0.116)
Log *CONSUMER DEPENDENTS COUNT*	-0.086***	-0.086***	-0.094***	-0.093***
	(0.027)	(0.027)	(0.030)	(0.030)
Log *SUPPLIER AGE*	0.021	0.019	-0.133	-0.130
	(0.227)	(0.226)	(0.292)	(0.289)
Log *SUPPLIER STARS COUNT*	0.112**	0.112**	0.119*	0.119*
	(0.055)	(0.056)	(0.065)	(0.065)
Log *SUPPLIER CONTRIBUTORS COUNT*	-0.239***	-0.238***	-0.253***	-0.254***
	(0.052)	(0.053)	(0.088)	(0.089)
Log *SUPPLIER DEPENDENTS COUNT*	-0.159***	-0.160***	-0.163***	-0.162***
	(0.033)	(0.034)	(0.031)	(0.030)
CRITICAL VULN	-0.068	-0.046	-0.137	-0.173*
	(0.071)	(0.097)	(0.091)	(0.091)
$S_{j,k}$ × *CRITICAL VULN*		-0.146		0.297
		(0.173)		(0.223)
CWE FEs	Yes	Yes	Yes	Yes
Consumer language FEs	Yes	Yes	Yes	Yes
Pseudo R-squared	0.328	0.328	0.312	0.312
Number of unique CWEs	75	75	81	81
Observations	28804650	28804650	31228011	31228011

Notes: Robust standard errors (clustered at FE variables) are shown in parentheses. The number of unique CWEs and observations are different from our main results because the set of valid CWEs (that is, appeared in the pre- and post-periods) changed as we modified the CWEs in the vulnerability disclosures from the suppliers/consumers. *** p < 0.01; ** p < 0.05; * p < 0.1

Similarity of Application Domains in Supplier and Consumer OSS

A related possibility is that OSS consumers may merely pay attention to and learn from other OSS projects in the same application domains. If OSS consumers and suppliers tend to belong to

the same domains, there is a risk that our results may conflate software dependencies with domain similarity. To investigate this alternative explanation that domain similarity may be driving the knowledge spillovers, we followed the recent software engineering literature that categorizes OSS projects and identifies similar OSS projects through the topics of their GitHub repositories (Izadi et al., 2021; Sharma et al., 2017). The rationale here is that if we could obtain topic labels for each of the supplier and consumer OSS in our sample, we can easily remove supplier–consumer dyads that have overlapped topics and reestimate our empirical models using the subsample, which presents no domain similarity in any supplier–consumer dyads.

There are two potential approaches to obtaining OSS topics. The first approach is to use the topic labels provided by the project owners. Since 2017, GitHub has allowed users to specify the topics of their own OSS repositories.[14] Given that project owners likely have the best knowledge about the nature and domain of their own projects, topic labels specified by them should provide the best characterization of their OSS. The issue with this approach is that many project owners have not specified topic labels for their OSS repositories. Therefore, another approach to obtaining topic labels is by analyzing the description of a project. On GitHub, project owners often (but not always) include a README file in their OSS repositories to communicate important information about their projects.[15] Prior studies have applied machine learning methods, such as topic modeling or multilabel classification, to predict topics from OSS projects' README files (e.g., Izadi et al., 2021; Sharma et al., 2017). These machine learning methods are helpful for deriving topic labels, especially when the project owners do not specify the topics of their GitHub repositories.

We consider both approaches in this analysis. In our original sample of 288,562 OSS

[14] https://github.blog/2017-01-31-introducing-topics/
[15] https://docs.github.com/en/repositories/managing-your-repositorys-settings-and-features/customizing-your-repository/about-readmes

projects, 175,384 (60.7%) had topic labels specified by their owners. Beyond these projects, we implemented the state-of-the-art machine learning procedure proposed by Izadi et al. (2021) to predict and obtain topic labels for additional 51,533 OSS projects that did not have owner-specified topics but had a README file with an English-based description (see Appendix A for details of this topic prediction procedure).

After removing supplier–consumer dyads that had overlapped topics, the number of dyads in our sample reduced from 385,722 in our original sample to 192,505 when considering only topic labels specified by the project owner and 234,539 when considering topic labels specified by the project owner and predicted from README files. Table 7 reports the empirical results estimated using these subsamples. Overall, these results are consistent with those in our main analysis and show similar patterns and magnitude for the coefficients of $S_{j,k}$ and the $S_{j,k} \times$ *CRITICAL VULN* interaction term. This suggests that domain similarity was not a plausible explanation for the observed knowledge spillovers, and consumers indeed learned from their suppliers rather than from OSS projects in the same application domains.

Table 7. Results from Supplier–Consumer Dyads without Overlapped Topics

Sources of topic labels	Specified by project owner		Specified by project owner and predicted from README file	
DV: $C_{i,k}$ (*CONSUMER DISCLOSES CWE k*)	(1) Full model	(2) Moderation	(3) Full model	(4) Moderation
$S_{j,k}$ (*SUPPLIER DISCLOSED CWE k*)	0.719***	0.636***	0.710***	0.628**
	(0.150)	(0.188)	(0.202)	(0.233)
Log *CONSUMER AGE*	-0.166	-0.166	-0.104	-0.105
	(0.346)	(0.346)	(0.279)	(0.279)
Log *CONSUMER STARS COUNT*	0.498***	0.498***	0.492***	0.492***
	(0.051)	(0.051)	(0.037)	(0.037)
Log *CONSUMER CONTRIBUTORS COUNT*	0.694***	0.691***	0.659***	0.657***
	(0.130)	(0.128)	(0.112)	(0.110)
Log *CONSUMER DEPENDENTS COUNT*	-0.107***	-0.105***	-0.099***	-0.098***
	(0.023)	(0.023)	(0.020)	(0.020)
Log *SUPPLIER AGE*	0.150	0.162	0.074	0.083
	(0.118)	(0.119)	(0.186)	(0.188)
Log *SUPPLIER STARS COUNT*	0.019	0.015	0.050	0.048
	(0.118)	(0.118)	(0.105)	(0.106)
Log *SUPPLIER CONTRIBUTORS COUNT*	-0.126	-0.120	-0.198*	-0.194
	(0.079)	(0.079)	(0.106)	(0.106)
Log *SUPPLIER DEPENDENTS COUNT*	-0.111*	-0.110*	-0.086	-0.085
	(0.055)	(0.054)	(0.052)	(0.051)
CRITICAL VULN	-0.050	-0.159*	-0.038	-0.134
	(0.077)	(0.086)	(0.098)	(0.112)
$S_{j,k}$ × *CRITICAL VULN*		0.450***		0.414**
		(0.105)		(0.149)
CWE FEs	Yes	Yes	Yes	Yes
Consumer language FEs	Yes	Yes	Yes	Yes
Pseudo R-squared	0.346	0.347	0.331	0.331
Number of unique CWEs	68	68	68	68
Observations	12915512	12915512	15729760	15729760

Notes: Robust standard errors (clustered at FE variables) are shown in parentheses. The number of unique CWEs and observations are different from our main results because the set of valid CWEs (that is, appeared in the pre- and post-periods) changed as we restricted this analysis to supplier–consumer dyads that had no overlapped topics. *** p < 0.01; ** p < 0.05; * p < 0.1

Knowledge Spillovers vs. Diffusion of Software Vulnerabilities

Another valid concern is that the vulnerability discovered in an OSS consumer may be the result of its supplier's vulnerability. In other words, what we observed could be merely the diffusion of

software vulnerabilities rather than knowledge transfer and organizational learning. Although a vulnerability does spread through software supply chains, it is important to note that the spread of a vulnerability will not lead to multiple CVE numbers. This is because only the upstream software in which the vulnerability originated should be assigned a CVE number.[16] As such, the vulnerabilities disclosed by the consumer OSS were unlikely to be the same as those disclosed by their upstream suppliers. Since we used CVE numbers to differentiate vulnerability disclosures and the vulnerability disclosures from the suppliers and consumers in our sample all had different CVE numbers, our empirical design should automatically avoid the threat that the supplier–consumer dyads were having the exact same software vulnerabilities. We nonetheless verified this potential threat by examining whether any of the vulnerability disclosures from the OSS consumers in our sample had mentioned or made a reference to their respective supplier's CVE number. As expected, we did not find any such instances.

Vulnerability Discoverers

We conducted a qualitative review and coding to address another alternative explanation: it may be some third-party OSS users or security researchers who learned from the suppliers' vulnerability disclosures and helped identify and report the vulnerabilities to the consumer OSS projects. In that case, we could still observe the spillover effect from vulnerability disclosures, but the mechanism would be different from organizational learning as we proposed. We investigated this alternative explanation by considering 30 randomly selected vulnerability disclosures from our sample's OSS consumers in the post-period. We manually reviewed their vulnerability disclosures on the NVD website. We traced back to their GitHub commits, issues, and pull requests to identify how the vulnerabilities were initially discovered, and more

[16] For CVE assignment rules, see https://www.cve.org/ResourcesSupport/AllResources/CNARules#section_7_assignment_rules.

importantly, who discovered and reported the vulnerabilities.

We labeled each of the vulnerability discoverers in one of the following categories: *core member*, *peripheral contributor*, or *security researcher*. Consistent with Crowston et al. (2006), a "core member" is a person affiliated with the OSS project or who makes repeated and numerous contributions in the development and maintenance of the software. A peripheral contributor, meanwhile, is an external software developer or user who is not affiliated with the project but makes infrequent or episodic contributions to the OSS project by reporting or fixing issues in the software (Barcomb et al., 2020). Finally, a security researcher is an external, independent actor who specializes in cybersecurity topics and practices.[17] To be able to label these discoverers, we systematically reviewed their GitHub user profiles, public code repositories, and patterns of engagement with the focal OSS project in which they discovered the vulnerability. The organizational learning mechanism is more likely if most of the vulnerabilities were discovered by the core members of the OSS projects. On the other hand, if most of the vulnerabilities were discovered by peripheral contributors or security researchers, our theorized mechanism based on knowledge transfer between an OSS supplier and its OSS consumer is less likely to be valid.

Two researchers worked independently on this qualitative exercise. The intercoder agreement from the initial round of coding was 0.9 (27/30; Cohen's kappa = 0.78; Cronbach's alpha = 0.84), suggesting a substantial degree of internal consistency. All disagreements were resolved in the second round. The results of this exercise are shown in Table B1 in Appendix B. Among the 30 vulnerabilities, we found that 20 were discovered by core members of the OSS projects, five by peripheral contributors, and five by security researchers. This means that

[17] OSS projects may have security experts on the team. For the purpose of this qualitative analysis, if the security expert is a member of the OSS project, we label them as a core member rather than a security researcher.

although peripheral contributors and security researchers can and will play a role in the discovery of software vulnerabilities in OSS projects, such external actors were unlikely to be the principal driver of the knowledge spillovers we observed in our main results because most of these vulnerabilities were discovered by project core members.

Learning by Contributing

It is common for OSS developers to affiliate with or contribute to multiple OSS projects. Therefore, it is possible that the core members who discovered the security vulnerabilities in their own projects were also contributors to their respective supplier projects. Consistent with the notions of "learning by contributing" (Nagle, 2018) and "learning from experience" (Boh et al., 2007), this approach would allow these OSS developers to gain firsthand knowledge about the security weaknesses in the supplier projects, and as a result, the supplier's vulnerability disclosures may not be necessary to initiate knowledge spillovers.

Given that supplier vulnerability disclosures are an essential pillar in our theoretical development, it is imperative for us to assess this alternative learning-by-contributing explanation. To this end, we investigated whether and how the 20 core members identified in our qualitative analysis contributed to their supplier projects. The results, reported in Table B2 in Appendix B, reveal that such instances are uncommon, although they did happen in a couple of these cases. This suggests that compared with the learning by contributing mechanism, the vulnerability disclosures from the suppliers were likely a more salient driver for the knowledge spillovers that we observed in our main results.

Sum of Evidence

In summary, our findings suggest a hitherto unexplored phenomenon of knowledge spillovers from supplier projects to consumer projects in the OSS ecosystem. These findings are robust to various alternative model specifications. Analyzing the underlying mechanisms, the evidence

shows that software dependencies are critical in effectuating knowledge spillovers, which are not explained by the domain similarity between supplier and consumer projects. Our qualitative analyses further reveal that the vulnerability discoverers were mostly the core members of the consumer OSS projects rather than third-party OSS developers or security researchers. Consistent with our theorizing, this suggests that security knowledge was indeed acquired by the consumer OSS project teams. We also found that most of these core members were not contributors to the supplier projects. This implies that members of consumer projects learned about security knowledge from their suppliers' public vulnerability disclosures rather than from direct experience working in the supplier projects. Taken together, we recognize and acknowledge that there could be multiple causal pathways leading to the observed knowledge spillovers (such as security researchers, peripheral contributors, and learning by contributing), and as a result, the true effect size might be smaller than the model suggests. Nevertheless, the empirical evidence collectively suggests that organizational learning from vulnerability disclosures through software supply chains was a more plausible and salient mechanism compared to the alternatives.

DISCUSSION AND CONCLUSION

OSS is an integral part of modern digital infrastructure. The growing number of OSS vulnerabilities and their adverse impacts on digital sovereignty are confronting us more than ever with the need to better understand and enhance open source security. Our research provides novel theoretical and empirical insights into the positive knowledge spillovers of vulnerability disclosures in the OSS ecosystem. That is, when an OSS project (i.e., a supplier) discloses a software vulnerability, the security knowledge will be transferred through software supply chains to downstream OSS projects (i.e., consumers) and enable the latter group to better identify new vulnerabilities with similar technical roots in their own code repositories. We further

demonstrate that knowledge spillover is moderated by the severity of supplier's vulnerability in which critical vulnerability, compared to noncritical vulnerability, yields nearly twice the effect size in inducing knowledge spillover.

Theoretical Contributions

This study makes three theoretical contributions. Our first contribution is to view open source security from the perspective of software supply chains. This perspective sheds light on a theoretically important relationship among OSS projects based on software dependencies, leading to the distinction between suppliers and consumers in the OSS ecosystem. Prior research has highlighted the issues of task interdependencies and developer interdependencies *within* an OSS project and explained how routines can be a coordination mechanism in OSS development (Lindberg et al., 2016). Our notion of software supply chains *across* OSS projects expands our understanding of interdependencies in OSS development from the task and developer levels to the artifact level. This gives rise to new opportunities to examine issues related to collaboration, learning, and knowledge sharing and reuse across OSS projects in their software supply chains.

Second, we developed a theory of open source security. Though OSS development and information security are both enduring topics in IS research, the intersection of the two has received hardly any direct attention in IS scholarship. We draw on organizational learning theory to explain why and how security knowledge in an OSS project's vulnerability disclosure can spill over to its downstream OSS consumers, enabling the latter to better discover software vulnerabilities with similar technical roots in their own projects (Mehrizi et al., 2022). The core of our theoretical development is the abstraction and codification of security weaknesses found in vulnerability disclosures and the transfer and reuse of security knowledge through the channel of software dependencies. We further suggest that knowledge spillover is moderated by the severity of the suppliers' vulnerabilities, whereby critical vulnerabilities would strengthen the

spillover effect. Our empirical analysis supports the proposed knowledge spillover and organizational learning through software supply chains, as well as the moderation role of the criticality of the supplier's vulnerability in the OSS ecosystem.

Finally, this research adds to the organizational learning literature. In developing our theoretical framework, we emphasized how OSS projects benefit from their suppliers. As illustrated in Table 8, our focus is on learning activities at the collective level in noncanonical organizations (i.e., online communities). This is an emerging and distinct form of organizational learning. The existing organizational learning literature tends to focus on learning in canonical and well-bounded organizations (i.e., corporations and institutions) at the individual and collective level (Majchrzak & Jarvenpaa, 2010; Owen-Smith & Powell, 2004) or learning in online communities at the individual level (Hwang et al., 2015). How one online community learns from another has received little direct attention in the organizational learning literature. We present one such case in the context of securing OSS projects.

Table 8. Taxonomy of Organizational Learning Research

		Organizational type	
		Canonical / bounded	**Noncanonical / unbounded**
Level of analysis	**Individual**	Knowledge sharing among individuals within or across corporations and institutions. *For example, homeland security professionals from different agencies share information to resolve security threats* (Majchrzak & Jarvenpaa, 2010).	Knowledge sharing among individuals in an online community. *For example, individuals in an online community share knowledge with others who have similar interests* (Hwang et al., 2015).
	Collective	Knowledge sharing among corporations and institutions. *For example, biotechnology firms in Boston improve their innovation performance through an industrial alliance* (Owen-Smith & Powell, 2004).	Knowledge sharing among different online communities. *For example, OSS projects share security knowledge to identify software vulnerabilities (this study).*

Practical Implication

Our findings regarding knowledge spillovers from vulnerability disclosures have important

practical and policy implications for open source security. While prior research focuses on the considerations of vulnerability disclosure for proprietary software vendors (Arora et al., 2010; Mitra & Ransbotham, 2015; Sen et al., 2020), our findings underscore the necessity of vulnerability disclosure for OSS projects and show the additional benefits of vulnerability disclosure to the security of software supply chains. In particular, an effective strategy to improve the security of OSS projects is to learn from their suppliers' vulnerabilities. From this perspective, OSS projects should be encouraged, even incentivized, to disclose their security vulnerabilities after the vulnerabilities are patched as part of their risk management routines (Germonprez et al., 2021). After OSS project teams discover and fix a security-related issue, a public disclosure about the existence of the vulnerability is beneficial to the security of the OSS ecosystem, even if the vulnerability has a lower severity. This is because by releasing an official vulnerability disclosure, the focal OSS project would be able to better retain the security knowledge from this experience through knowledge abstraction and codification. In addition, it will facilitate downstream OSS consumers in identifying security weaknesses of the same nature in their projects.

Limitations and Future Research

Our work has limitations. First, aside from the criticality of the vulnerability, we did not consider other contingencies that may affect knowledge spillover from OSS projects' vulnerability disclosures. This is to maintain the focus and parsimony of our theory (Weber, 2003). Nevertheless, it is likely that there are other factors at the levels of the platform, OSS project, or individual developers that can facilitate or hinder the degree of knowledge spillovers in our setting. Future research could extend our theoretical framework, with consideration of these contextual factors to broaden the understanding of the contingencies in knowledge spillovers and their boundary conditions.

Second, we were not able to examine the temporal dynamics of learning in OSS projects owing to the nature of our cross-sectional data. As Argote et al. (2021) pointed out, "organizations vary in the rate at which they learn" (p. 5399). OSS projects may have different learning curve patterns based on their projects' complexity, maturity, and popularity. Therefore, future researchers with access to panel data could extend our theoretical framework to explain the temporal dynamics of the spillover of security knowledge in vulnerability disclosures.

Third, for conceptual simplicity and computational feasibility, we restricted our theoretical development and empirical analysis to the knowledge transfer from suppliers to their direct consumers and unversioned software dependencies. However, a case can be made that the knowledge transfer could go beyond one level of dependencies and reach all downstream OSS projects. Similarly, the consideration of versioned software dependencies could potentially reveal more insights into the nature and history of the supplier–consumer relationship. We welcome future research that may relax these restrictions.

Finally, we limited our empirical analysis to a sample of OSS projects on GitHub to alleviate platform-specific heterogeneity and unobservables. Although GitHub is by far the most popular platform for OSS development and has been used in much OSS research (e.g., Chen et al., 2022; Lin & Maruping, 2022; Lindberg et al., 2016), there exist many other similar platforms, such as Bitbucket, GitLab, and SourceForge. Future investigators may consider OSS projects on other (or multiple) platforms to examine the generalizability of our theory.

REFERENCES

Ahmad, A., Desouza, K. C., Maynard, S. B., Naseer, H., & Baskerville, R. L. (2020). How integration of cyber security management and incident response enables organizational learning. *Journal of the Association for Information Science and Technology*, *71*(8), 939–953.

Ahmed, A., & Lee, B. (2020). Organizational learning on bug bounty platforms. *AMCIS 2020 Proceedings*. https://aisel.aisnet.org/amcis2020/info_security_privacy/info_security_privacy/33

Alberts, C. J., Dorofee, A. J., Creel, R., Ellison, R. J., & Woody, C. (2011). A systemic approach for assessing software supply-chain risk. *2011 44th Hawaii International Conference on System Sciences*, 1–8.

Altinkemer, K., Rees, J., & Sridhar, S. (2008). Vulnerabilities and patches of open source software: An empirical study. *Journal of Information System Security*, *4*(2), 3–25.

Argote, L. (2013). *Organizational Learning: Creating, Retaining and Transferring Knowledge* (2nd ed.). Springer.

Argote, L., Lee, S., & Park, J. (2021). Organizational learning processes and outcomes: Major findings and future research directions. *Management Science*, *67*(9), 5399–5429.

Arora, A., Krishnan, R., Telang, R., & Yang, Y. (2010). An empirical analysis of software vendors' patch release behavior: Impact of vulnerability disclosure. *Information Systems Research*, *21*(1), 115–132.

Arora, A., & Telang, R. (2005). Economics of software vulnerability disclosure. *IEEE Security & Privacy*, *3*(1), 20–25.

Arora, A., Telang, R., & Xu, H. (2008). Optimal policy for software vulnerability disclosure. *Management Science*, *54*(4), 642–656.

Autio, E., Hameri, A.-P., & Vuola, O. (2004). A framework of industrial knowledge spillovers in big-science centers. *Research Policy*, *33*(1), 107–126.

Barcomb, A., Kaufmann, A., Riehle, D., Stol, K.-J., & Fitzgerald, B. (2020). Uncovering the periphery: A qualitative survey of episodic volunteering in free/libre and open source software communities. *IEEE Transactions on Software Engineering*, *46*(9), 962–980.

Berlin Declaration. (2020). *Berlin declaration on digital society and value-based digital government*. https://www.bmi.bund.de/SharedDocs/pressemitteilungen/EN/2020/12/berlin-declaration-digitalization.html

Bogart, C., Kästner, C., & Herbsleb, J. (2015). When it breaks, it breaks: How ecosystem developers reason about the stability of dependencies. *2015 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, 86–89.

Boh, W. F. (2007). Mechanisms for sharing knowledge in project-based organizations. *Information and Organization*, *17*(1), 27–58.

Boh, W. F., Slaughter, S. A., & Espinosa, J. A. (2007). Learning from Experience in Software Development: A Multilevel Analysis. *Management Science*, *53*(8), 1315–1331.

Cavusoglu, H., Cavusoglu, H., & Raghunathan, S. (2007). Efficiency of vulnerability disclosure mechanisms to disseminate vulnerability knowledge. *IEEE Transactions on Software Engineering*, *33*(3), 171–185.

Chen, W., Jin, F., & Xue, L. (2022). Flourish or perish? The impact of technological acquisitions on contributions to open-source software. *Information Systems Research*, *33*(3), 867–886.

Chinthanet, B., Kula, R. G., McIntosh, S., Ishio, T., Ihara, A., & Matsumoto, K. (2021). Lags in the release, adoption, and propagation of npm vulnerability fixes. *Empirical Software Engineering*, *26*(3), 47.

Cox, R. (2019). Surviving software dependencies. *Communications of the ACM*, *62*(9), 36–43.

Crowston, K., Wei, K., Li, Q., & Howison, J. (2006). Core and periphery in free/libre and open source software team communications. *Proceedings of the 39th Annual Hawaii International Conference on System Sciences (HICSS'06)*, *6*, 118a–118a.

Dahlin, K. B., Chuang, Y.-T., & Roulet, T. J. (2018). Opportunity, motivation, and ability to learn from failures and errors: Review, synthesis, and ways to move forward. *Academy of Management Annals*, *12*(1), 252–277.

Daniel, S., Maruping, L., Cataldo, M., & Herbsleb, J. (2018). The impact of ideology misfit on open source software communities and companies. *MIS Quarterly*, *42*(4), 1069–1096.

Decan, A., Mens, T., & Grosjean, P. (2019). An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering*, *24*(1), 381–416.

Drupsteen, L., & Guldenmund, F. W. (2014). What is learning? A review of the safety literature to define learning from incidents, accidents and disasters. *Journal of Contingencies and Crisis Management*, *22*(2), 81–96.

Eghbal, N. (2020). *Working in Public: The Making and Maintenance of Open Source Software*. Stripe Press.

Ellison, R. J., & Woody, C. (2010). Supply-chain risk management: Incorporating security into software development. *43rd Hawaii International Conference on System Sciences*, 1–10.

Executive Office of the President. (2021). *Executive Order 14028 on Improving the Nation's Cybersecurity*. https://www.federalregister.gov/documents/2021/05/17/2021-10460/improving-the-nations-cybersecurity

Faraj, S., Jarvenpaa, S. L., & Majchrzak, A. (2011). Knowledge collaboration in online communities. *Organization Science*, *22*(5), 1224–1239.

Ferraiuolo, A., Behjati, R., Santoro, T., & Laurie, B. (2022). Policy transparency: Authorization logic meets general transparency to prove software supply chain integrity. *Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*, 3–13.

Frese, M., & Keith, N. (2015). Action errors, error management, and learning in organizations. *Annual Review of Psychology*, *66*(1), 661–687.

Gal-Or, E., & Ghose, A. (2005). The economic incentives for sharing security information. *Information Systems Research*, *16*(2), 186–208.

Geiger, R. S., Howard, D., & Irani, L. (2021). The labor of maintaining and scaling free and open-source software projects. *Proceedings of the ACM on Human-Computer Interaction*, *5*(CSCW1), 175:1-175:28.

Germonprez, M., Gandhi, R., & Link, G. (2021). The routinization of open source project engagement: The case of open source risk management routines. *Communications of the Association for Information Systems*, *49*(1).

Germonprez, M., Kendall, J. E., Kendall, K. E., Mathiassen, L., Young, B., & Warner, B. (2017). A theory of responsive design: A field study of corporate engagement with open source communities. *Information Systems Research*, *28*(1), 64–83.

Goggins, S., Lumbard, K., & Germonprez, M. (2021). Open source community health: Analytical metrics and their corresponding narratives. *2021 IEEE/ACM 4th International Workshop on Software Health in Projects, Ecosystems and Communities (SoHeal)*, 25–33.

Haefliger, S., von Krogh, G., & Spaeth, S. (2008). Code reuse in open source software. *Management Science*, *54*(1), 180–193.

Hansen, M. T. (1999). The search-transfer problem: The role of weak ties in sharing knowledge across organization subunits. *Administrative Science Quarterly*, *44*(1), 82–111.

Homsma, G. J., Van Dyck, C., De Gilder, D., Koopman, P. L., & Elfring, T. (2009). Learning from error: The influence of error incident characteristics. *Journal of Business Research*, *62*(1), 115–122.

Howison, J., & Crowston, K. (2014). Collaboration through open superposition: A theory of the open source way. *MIS Quarterly*, *38*(1), 29–50.

Hwang, E. H., Singh, P. V., & Argote, L. (2015). Knowledge sharing in online communities: Learning to cross geographic and hierarchical boundaries. *Organization Science*, *26*(6), 1593–1611.
Iacus, S. M., King, G., & Porro, G. (2011). Multivariate matching methods that are monotonic imbalance bounding. *Journal of the American Statistical Association*, *106*(493), 345–361.
Iacus, S. M., King, G., & Porro, G. (2012). Causal inference without balance checking: Coarsened exact matching. *Political Analysis*, *20*(1), 1–24.
Izadi, M., Heydarnoori, A., & Gousios, G. (2021). Topic recommendation for software repositories using multi-label classification algorithms. *Empirical Software Engineering*, *26*(5), 93.
Jacobides, M. G., Cennamo, C., & Gawer, A. (2018). Towards a theory of ecosystems. *Strategic Management Journal*, *39*(8), 2255–2276.
Katz, J. (2020). *Libraries.io Open Source Repository and Dependency Metadata*. Zenodo. https://zenodo.org/record/3626071
Kula, R. G., German, D. M., Ouni, A., Ishio, T., & Inoue, K. (2018). Do developers update their library dependencies? *Empirical Software Engineering*, *23*(1), 384–417.
Lifshitz-Assaf, H., & Nagle, F. (2021). The digital economy runs on open source. Here's how to protect it. *Harvard Business Review*, 1–7.
Lin, Y.-K., & Maruping, L. M. (2022). Open source collaboration in digital entrepreneurship. *Organization Science*, *33*(1), 212–230.
Lindberg, A., Berente, N., Gaskin, J., & Lyytinen, K. (2016). Coordinating interdependencies in online communities: A study of an open source software project. *Information Systems Research*, *27*(4), 751–772.
MacCormack, A., Rusnak, J., & Baldwin, C. Y. (2006). Exploring the structure of complex software designs: An empirical study of open source and proprietary code. *Management Science*, *52*(7), 1015–1030.
Madsen, P. M., & Desai, V. (2010). Failing to learn? The effects of failure and success on organizational learning in the global orbital launch vehicle industry. *Academy of Management Journal*, *53*(3), 451–476.
Majchrzak, A., & Jarvenpaa, S. L. (2010). Safe contexts for interorganizational collaborations among homeland security professionals. *Journal of Management Information Systems*, *27*(2), 55–86.
Malhotra, A., Majchrzak, A., & Lyytinen, K. (2021). Socio-technical affordances for large-scale collaborations: Introduction to a virtual special issue. *Organization Science*, *32*(5), 1371–1390.
McLaughlin, M.-D., & Gogan, J. (2018). Challenges and best practices in information security management. *MIS Quarterly Executive*, *17*(3). https://aisel.aisnet.org/misqe/vol17/iss3/6
Mehrizi, M. H. R., Nicolini, D., & Modol, J. (2022). How do organizations learn from information system incidents? A synthesis of the past, present, and future. *MIS Quarterly*, *46*(1), 531–590.
Mitra, S., & Ransbotham, S. (2015). Information disclosure and the diffusion of information security attacks. *Information Systems Research*, *26*(3), 565–584.
Nagle, F. (2018). Learning by contributing: Gaining competitive advantage through contribution to crowdsourced public goods. *Organization Science*, *29*(4), 569–587.
Nagle, F. (2019). Open source software and firm productivity. *Management Science*, *65*(3), 1191–1215.
Nagle, F., Dana, J., Hoffman, J., Randazzo, S., & Zhou, Y. (2022). *Census II of Free and Open*

Source Software—Application Libraries. The Linux Foundation and The Laboratory for Innovation Science at Harvard. https://linuxfoundation.org/wp-content/uploads/LFResearch_Harvard_Census_II.pdf

Nambisan, S., Lyytinen, K., Majchrzak, A., & Song, M. (2017). Digital innovation management: Reinventing innovation management research in a digital world. *MIS Quarterly*, *41*(1), 223–238.

Nonaka, I., & von Krogh, G. (2009). Tacit knowledge and knowledge conversion: Controversy and advancement in organizational knowledge creation theory. *Organization Science*, *20*(3), 635–652.

O'Mahony, S., & Ferraro, F. (2007). The emergence of governance in an open source community. *Academy of Management Journal*, *50*(5), 1079–1106.

Owen-Smith, J., & Powell, W. W. (2004). Knowledge networks as channels and conduits: The effects of spillovers in the boston biotechnology community. *Organization Science*, *15*(1), 5–21.

Pashchenko, I., Vu, D.-L., & Massacci, F. (2020). A qualitative study of dependency management and its security implications. *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 1513–1531.

Payne, C. (2002). On the security of open source software. *Information Systems Journal*, *12*(1), 61–78.

Prana, G. A. A., Sharma, A., Shar, L. K., Foo, D., Santosa, A. E., Sharma, A., & Lo, D. (2021). Out of sight, out of mind? How vulnerable dependencies affect open-source projects. *Empirical Software Engineering*, *26*(4), 1–34.

Ransbotham, S. (2010). An empirical analysis of exploitation attempts based on vulnerabilities in open source software. *Proceedings of the 9th Workshop on the Economics of Information Security*. Workshop on the Economics of Information Security (WEIS), Cambridge, MA.

Ransbotham, S., Mitra, S., & Ramsey, J. (2012). Are markets for vulnerabilities effective? *MIS Quarterly*, *36*(1), 43–64.

Roberts, J. A., Il-Horn Hann, & Slaughter, S. A. (2006). Understanding the motivations, participation, and performance of open source software developers: A longitudinal study of the apache projects. *Management Science*, *52*(7), 984–999. aqh.

Safadi, H., Johnson, S. L., & Faraj, S. (2021). Who contributes knowledge? Core-periphery tension in online innovation communities. *Organization Science*, *32*(3), 752–775.

Schryen, G. (2011). Is open source security a myth? *Communications of the ACM*, *54*(5), 130–140.

Sen, R., Choobineh, J., & Kumar, S. (2020). Determinants of software vulnerability disclosure timing. *Production and Operations Management*, *29*(11), 2532–2552.

Sharma, A., Thung, F., Kochhar, P. S., Sulistya, A., & Lo, D. (2017). Cataloging GitHub repositories. *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*, 314–319.

Skopik, F., Settanni, G., & Fiedler, R. (2016). A problem shared is a problem halved: A survey on the dimensions of collective cyber defense through security information sharing. *Computers & Security*, *60*, 154–176.

Spaeth, S., von Krogh, G., & He, F. (2015). Perceived firm attributes and intrinsic motivation in sponsored open source software projects. *Information Systems Research*, *26*(1), 224–237.

Stanko, M. A. (2016). Toward a theory of remixing in online innovation communities. *Information Systems Research*, *27*(4), 773–791.

Synopsys. (2022). *Open Source Security and Risk Analysis Report*.

https://www.synopsys.com/software-integrity/resources/analyst-reports/open-source-security-risk-analysis.html
Temizkan, O., Kumar, R. L., Park, S., & Subramaniam, C. (2012). Patch release behaviors of software vendors in response to vulnerabilities: An empirical analysis. *Journal of Management Information Systems*, *28*(4), 305–338.
The Linux Foundation. (2022). *Addressing Cybersecurity Challenges in Open Source Software*. https://linuxfoundation.org/tools/addressing-cybersecurity-challenges-in-open-source-software/
Tullio, D. D., & Staples, D. S. (2013). The governance and control of open source software projects. *Journal of Management Information Systems*, *30*(3), 49–80.
von Krogh, G., Haefliger, S., Spaeth, S., & Wallin, M. W. (2012). Carrots and rainbows: Motivation and social practice in open source software development. *MIS Quarterly*, *36*(2), 649–676.
Weber, R. (2003). Theoretically speaking. *MIS Quarterly*, *27*(3), iii–xii.
Yoo, Y., Henfridsson, O., & Lyytinen, K. (2010). The new organizing logic of digital innovation: An agenda for information systems research. *Information Systems Research*, *21*(4), 724–735.
Zander, U., & Kogut, B. (1995). Knowledge and the speed of the transfer and imitation of organizational capabilities: An empirical test. *Organization Science*, *6*(1), 76–92.
Zollo, M., & Reuer, J. J. (2010). Experience spillovers across corporate development activities. *Organization Science*, *21*(6), 1195–1212.
Zollo, M., & Winter, S. G. (2002). Deliberate learning and the evolution of dynamic capabilities. *Organization Science*, *13*(3), 339–351.

APPENDIX A

Predicting Topics for GitHub Repositories

According to GitHub, topics are labels that create subject-based connections between GitHub repositories. As such, topics can be useful for identifying the application domains of OSS projects. For example, the maintainers of Apache Log4j specify the project to have the following topic labels: java, api, library, log4j, jvm, logger, logging, syslog, apache, and log4j2. Although topics can help improve the discoverability and reveal important qualities of GitHub projects, many GitHub projects do not have any topic labels assigned to them because doing so requires additional effort and attention from the project owners. Therefore, several researchers have proposed machine learning methods for predicting topics for GitHub repositories. To make topic label predictions, scholars have utilized bytecode and interdependencies (Vargas-Baldrich et al., 2015), domain knowledge from StackOverflow (Cai et al., 2016), and textual descriptions (Izadi et al., 2021).

Given the characteristics of our data, the topic prediction approach proposed by Izadi et al. (2021) is most applicable. The approach consists of three steps: data preparation and preprocessing; model training; and model evaluation and use. We followed their approach to predict the topics of OSS projects in our sample in situations where the projects do not have any owner-specified topic labels. Technical details of this approach can be found in Izadi et al. (2021). In what follows, we provide a high-level overview of implementing this approach in our setting.

Step 1: data preparation and preprocessing. As in any supervised machine learning exercise, the first step is to develop a testbed of instances with appropriate labels. We identified and downloaded GitHub projects that have an English-based README file and have been assigned topic labels by their project owners. We preprocessed the textual descriptions in

README files using common text mining procedures, which include tokenizing the text; converting tokens to lowercase; removing punctuations, digits, and URLs; omitting stop words; and removing tokens with a frequency of fewer than 50 in the collection. The remaining tokens in a README description served as features for the respective OSS project. The topic labels are the targets of our predictions. The topics were matched against GitHub's featured topics or their aliases. Although over 5000 unique topics were initially identified, we followed Izadi et al. (2021) and focused on predicting the GitHub-featured topics because these topics are validated and curated by the entire GitHub community. We further restricted the topic labels to the top 200 most frequent topics because they cumulatively represented over 90% of topics being assigned to GitHub repositories. At the end, our testbed comprised a collection of 130,869 GitHub projects with English textual descriptions and at least one GitHub-featured topic. We split these GitHub projects into a training set (80%) and a testing set (20%).

Step 2: model training. Using data from the training set, we trained multiple text classifiers to learn the relationship between tokens in README descriptions and their corresponding topics. Because an OSS project can have multiple topics, the text classifiers were set up to perform multilabel classification. Izadi et al. (2021) considered four candidate text classifiers. The first classifier is Naive Bayes (denoted by NB). This classifier can be implemented in two ways: one based on the term frequency-inverse document frequency (TF-IDF) features, and another using a Doc2Vec representation in which word-embedding vectors were used to capture the semantic meanings of textual descriptions. The second classifier is logistic regression (denoted by LR), which learned a function to predict the log odds of each topic given the TF-IDF features or the Doc2Vec representations of the textual descriptions. The third classifier is FastText, which learned the Word2Vec representations of words in the textual descriptions of GitHub projects and used a hierarchical softmax loss function to train a tree-

based classifier. The last classifier is based on DistilBERT, which is the condensed version of the state-of-the-art BERT model via an introduction of knowledge distillation during the pretraining phase. DistilBERT retains 97% of BERT's language understanding capability with 60% faster training time. A multilabel classification layer is added on top of DistilBERT for fine-tuning the model for topic prediction. Additionally, instances with less frequent topics are assigned higher weights in DistilBERT's loss function to address the imbalanced topic distribution in the training data. The parameters of the classifiers followed the configurations in Izadi et al. (2021). For example, the learning rate of DistilBERT was set to 3e-5, the maximum input length to 512, and the batch size to four. The number of features for TF-IDF was set to 20,000, and the dimension of Doc2Vec was set to 1,000 with a minimum frequency of 10.

Step 3: model evaluation and use. We followed Izadi et al. (2021) and evaluated the performance of the candidate text classifiers using five evaluation metrics: recall, precision, F1 measure, success rate, and label ranking average precision (LRAP). Recall quantifies the percentage of actual topics that are correctly predicted. Recall-at-5 (denoted by R@5), hence, is the average percentage of actual topics that are correctly predicted in the model's top five predicted topic labels. Precision is measured by the percentage of predicted topics that are correct. Precision-at-5 (denoted by P@5) reports the average percentage of correct topic predictions in the model's top five predicted topic labels. F1 measure-at-5 (denoted by F@5) is the harmonic mean of recall-at-5 and precision-at-5. The success rate at k (denoted by S) measures the percentage of each model's top k predicted topics that are correct. S@1 measures whether the most probable topic predicted by the model is correct. S@5 measures whether at least one of the five most probable topics predicted by the model is correct. LRAP examines the ranking of the probable topics predicted by each model and computes the overall percentage of higher-ranked topics that are correct. LRAP is calculated by the average label ranking precision

of each project, which is calculated by $\sum_{j \in J}(higher_j/rank_j)/|J|$, where J is the set of correct topics, $rank_j$ is the rank of topic j, and $higher_j$ is the number of correct topics ranked higher than topic j. Table A1 summarizes the performance of the candidate models when evaluated against data from the testing set. We found that DistilBERT outperformed other models across all evaluation metrics, which is consistent with the findings reported by Izadi et al. (2021). Hence, we chose to leverage the DistilBERT model to predict the topic labels of OSS projects in our sample when they were not specified by their project owners.

Table A1. Performance Evaluation Results

Model	S@1	S@5	R@5	P@5	F1@5	LRAP
NB, D2V	0.218	0.629	0.459	0.167	0.232	0.171
NB, TF-IDF	0.320	0.421	0.292	0.100	0.140	0.220
LR, D2V	0.195	0.620	0.445	0.159	0.222	0.154
LR, TF-IDF	0.373	0.854	0.705	0.256	0.356	0.320
FastText	0.634	0.873	0.725	0.263	0.363	0.630
DistilBERT	0.651	0.900	0.770	0.284	0.392	0.665

APPENDIX B

Detailed Results of Qualitative Review and Coding

Table B1. Analyzing Discoverers of OSS Consumers' Vulnerabilities

#	OSS consumer	CVE number	Discoverer	Discoverer type
1	23andme/yamale	CVE-2021-38305	mildebrandt	Core member
2	abel533/Mapper	CVE-2022-36594	sybb0743	Security researcher
3	alerta/alerta	CVE-2020-26214	satterly	Core member
4	authelia/authelia	CVE-2021-29456	james-d-elliott	Core member
5	bolt/core	CVE-2021-27367	bobdenotter	Core member
6	clientIO/joint	CVE-2021-23444	kumilingus	Core member
7	codecov/codecov-node	CVE-2020-7597	drazisil	Core member
8	doctrine/dbal	CVE-2021-43608	morozov	Security researcher
9	dpgaspar/flask-appbuilder	CVE-2021-29621	dpgaspar	Core member
10	dropwizard/dropwizard	CVE-2020-11002	joschi	Core member
11	eclipse-theia/theia	CVE-2021-28161	luigigubello	Core member
12	ethercreative/logs	CVE-2021-32752	Tam	Core member
13	facebook/hermes	CVE-2021-24037	dulinriley	Core member
14	forkcms/forkcms	CVE-2020-23960	carakas	Core member
15	formstone/formstone	CVE-2020-26768	adrianomarcmont	Security researcher
16	http4s/blaze	CVE-2021-21293	rossabaker	Peripheral contributor
17	kaminari/kaminari	CVE-2020-11082	viseztrance	Core member
18	locka99/opcua	CVE-2022-25903	locka99	Core member
19	mermaid-js/mermaid	CVE-2021-43861	knsv	Peripheral contributor
20	mithunsatheesh/node-rules	CVE-2020-7609	mithunsatheesh	Core member
21	mjmlio/mjml	CVE-2020-12827	kmcb777	Core member
22	ome/omero-web	CVE-2021-41132	Lachlan Horsey	Security researcher
23	prismjs/prism	CVE-2021-3801	ready-research	Peripheral contributor
24	publify/publify	CVE-2021-25974	mvz	Peripheral contributor
25	rare-technologies/bounter	CVE-2021-41497	Daybreak2019	Security researcher
26	reg-viz/reg-suit	CVE-2021-32673	progfay	Core member
27	tauri-apps/tauri	CVE-2022-39215	martin-ocasek	Security researcher
28	tryghost/ghost	CVE-2021-39192	zn9988	Peripheral contributor
29	vyperlang/vyper	CVE-2021-41122	charles-cooper	Core member
30	yahoo/elide	CVE-2020-5289	wcekan	Core member

Table B2. Core-Member Discoverers in Consumer Projects and Their Contributions to Supplier Projects

Consumer project	Vulnerability discoverer (core member in the consumer project)	Supplier project	Discoverer's contributions to supplier project
23andme/yamale	mildebrandt	yaml/pyyaml	None
alerta/alerta	satterly	pyca/cryptography	None
		pallets/flask	None
		yaml/pyyaml	None
authelia/authelia	james-d-elliott	mde/ejs	None
		twbs/bootstrap	None
		auth0/node-jsonwebtoken	None
		jquery/jquery	None
		request/request	None
		unshiftio/url-parse	None
bolt/core	bobdenotter	sebastianbergmann/phpunit	None
		twigphp/twig	None
		erusev/parsedown	None
clientio/joint	kumilingus	jashkenas/backbone	None
		jquery/jquery	None
		lodash/lodash	None
		webpack/webpack-dev-server	None
codecov/codecov-node	drazisil	request/request	None
doctrine/dbal	morozov	sebastianbergmann/phpunit	10 commits
dpgaspar/flask-appbuilder	dpgaspar	pallets/flask	None
dropwizard/dropwizard	joschi	eclipse/jetty.project	3 commits
		fasterxml/jackson-databind	None
		scala/scala	None
ethercreative/logs	Tam	craftcms/cms	None
facebook/hermes	dulinriley	request/request	None
		npm/node-tar	None
forkcms/forkcms	carakas	symfony/symfony	None
		sebastianbergmann/phpunit	None
http4s/blaze	rossabaker	asynchttpclient/async-http-client	None
		scala/scala	None
locka99/opcua	locka99	dtolnay/serde-yaml	None
mermaid-js/mermaid	knsv	lodash/lodash	None
		moment/moment	None
		webpack/webpack-dev-server	None
		dominictarr/event-stream	None
mithunsatheesh/node-rules	mithunsatheesh	lodash/lodash	None
mjmlio/mjml	kmcb777	lodash/lodash	None
		jquery/jquery	None
publify/publify	mvz	flori/json	None
vyperlang/vyper	charles-cooper	ethereum/py-evm	None
yahoo/elide	wcekan	fasterxml/jackson-databind	None
		eclipse/jetty.project	None