# Research Directions in Software Supply Chain Security

LAURIE WILLIAMS, North Carolina State University, Raleigh, North Carolina, USA
GIACOMO BENEDETTI, University of Genoa, Genoa, Italy
SIVANA HAMER, North Carolina State University, Raleigh, North Carolina, USA
RANINDYA PARAMITHA, University of Trento, Trento, Italy
IMRANUR RAHMAN, MAHZABIN TAMANNA, GREG TYSTAHL, NUSRAT ZAHAN, and PATRICK MORRISON, North Carolina State University, Raleigh, North Carolina, USA
YASEMIN ACAR, Paderborn University, Paderborn, Germany
MICHEL CUKIER, University of Maryland, College Park, Maryland, USA
CHRISTIAN KÄSTNER, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA
ALEXANDROS KAPRAVELOS, DOMINIK WERMKE, and WILLIAM ENCK, North Carolina State University, Raleigh, North Carolina, USA

Reusable software libraries, frameworks, and components, such as those provided by open source ecosystems and third-party suppliers, accelerate digital innovation. However, recent years have shown almost exponential growth in attackers leveraging these software artifacts to launch software supply chain attacks. Past well-known software supply chain attacks include the SolarWinds, log4j, and xz utils incidents. Supply chain attacks are considered to have three major attack vectors: through vulnerabilities and malware accidentally or intentionally injected into open source and third-party *dependencies/components/containers*; by infiltrating the *build infrastructure* during the build and deployment processes; and through targeted techniques aimed at the *humans* involved in software development, such as through social engineering. Plummeting trust in the software supply chain could decelerate digital innovation if the software industry reduces its use of open source and third-party artifacts to reduce risks. This article contains perspectives and knowledge obtained from intentional outreach with practitioners to understand their practical challenges and from extensive

This material is based upon work supported by the National Science Foundation (Grant Nos. 2207008, 2206859, 2206865, and 2206921). Any opinions expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. R. Paramitha is partly supported by the European Union's Horizon 2020 research and innovation program under grant agreement number 101120393 (Sec4AI4Sec).
Authors' Contact Information: Laurie Williams (corresponding author), North Carolina State University, Raleigh, North Carolina, USA; e-mail: lawilli3@ncsu.edu; Giacomo Benedetti, University of Genoa, Genoa, Italy; e-mail: giacomo.benedetti@dibris.unige.it; Sivana Hamer, North Carolina State University, Raleigh, North Carolina, USA; e-mail: sahamer@ncsu.edu; Ranindya Paramitha, DISI, University of Trento, Trento, Italy; e-mail: ranindya.paramitha@unitn.it; Imranur Rahman, North Carolina State University, Raleigh, North Carolina, USA; e-mail: irahman3@ncsu.edu; Mahzabin Tamanna, North Carolina State University, Raleigh, North Carolina, USA; e-mail: mtamann@ncsu.edu; Greg Tystahl, North Carolina State University, Raleigh, North Carolina, USA; e-mail: gttystah@ncsu.edu; Nusrat Zahan, North Carolina State University, Raleigh, North Carolina, USA; e-mail: nzahan@ncsu.edu; Patrick Morrison, North Carolina State University, Raleigh, North Carolina, USA; e-mail: pjmorris@ncsu.edu; Yasemin Acar, Paderborn University, Paderborn, Germany; e-mail: acar@sec.uni-hannover.de; Michel Cukier, University of Maryland, College Park, Maryland, USA; e-mail: mcukier@umd.edu; Christian Kästner, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA; e-mail: kaestner@cs.cmu.edu; Alexandros Kapravelos, North Carolina State University, Raleigh, North Carolina, USA; e-mail: akaprav@ncsu.edu; Dominik Wermke, North Carolina State University, Raleigh, North Carolina, USA; e-mail: dwermke@ncsu.edu; William Enck, North Carolina State University, Raleigh, North Carolina, USA; e-mail: whenck@ncsu.edu.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM 1557-7392/2025/5-ART146
https://doi.org/10.1145/3714464

research efforts. We then provide an overview of current research efforts to secure the software supply chain. Finally, we propose a future research agenda to close software supply chain attack vectors and support the software industry.

CCS Concepts: • **Security and privacy → Software security engineering**;

Additional Key Words and Phrases: Software security, Software supply chain security, Open source security

**ACM Reference format:**
Laurie Williams, Giacomo Benedetti, Sivana Hamer, Ranindya Paramitha, Imranur Rahman, Mahzabin Tamanna, Greg Tystahl, Nusrat Zahan, Patrick Morrison, Yasemin Acar, Michel Cukier, Christian Kästner, Alexandros Kapravelos, Dominik Wermke, and William Enck. 2025. Research Directions in Software Supply Chain Security. *ACM Trans. Softw. Eng. Methodol.* 34, 5, Article 146 (May 2025), 38 pages.
https://doi.org/10.1145/3714464

## 1 Introduction

The modern world relies on digital innovation in almost every human endeavor, including critical infrastructure. Digital innovation has accelerated substantially as software is increasingly built with layers of reusable abstractions, including libraries, frameworks, cloud infrastructure, and AI modules. This style of software development is considered a *software supply chain* where software projects depend on and build upon other software projects. Based upon its analysis of 1,067 commercial code bases across 17 industries [215], Synopsys reports that 96% of code bases contained open source software and 77% of the code in the code bases were open source software.

Software developers did not anticipate how the software supply chain would become a deliberate attack vector. The software industry has moved from passive adversaries finding and exploiting vulnerabilities contributed by honest, well-intentioned developers to a new generation of software supply chain attacks that target developers directly, bypassing existing defenses. Attackers implant vulnerabilities directly into open source software components and infect build and deployment pipelines. In 2024, Sonatype reported that 512,847 malicious open source packages were logged, reflecting a 156% year-over-year increase [206]. Well-known attacks, such as SolarWinds [60, 204], log4j [7], and xz utils [64, 84], affected thousands of customers and hundreds of businesses and government agencies throughout the world. Consequently, the US **Executive Order (EO)** 14028 [92] and the European Cybersecurity Resilience Act [56] highlight the urgent need to improve transparency and integrity in open source artifacts to enhance supply chain security.

As shown in Figure 1, software supply chain attacks are considered to have three major attack vectors. The first is through vulnerabilities and malware accidentally or maliciously injected into open source and third-party *code dependencies*, such as components, containers, and AI models. This attack vector appears in the lower "component" layer in the figure. For example, log4j [7] is an example of an accidentally injected vulnerability. However, once discovered, attackers exploited this vulnerability to execute arbitrary code, steal sensitive data like passwords, install ransomware, establish backdoors, mine cryptocurrency, and take control of affected networks. In what is referred to as "protestware," an attacker injected malicious code into npm package node-ipc [1] at the start of the Russian-Ukraine conflict in 2022. Malicious components are injected into ecosystems, such as npm, PyPI, and Maven, so that they are downloaded by applications, enabling attackers to exploit. The malicious code was intended to overwrite arbitrary files dependent upon the geo-location of the user IP address. Based upon an analysis of real-world attacks and scientific and grey literature, Ladisa et al. [117] contributed a taxonomy of open source software supply chain attacks that have the goal of injecting malicious code into open source components. We discuss this attack vector in Section 3.

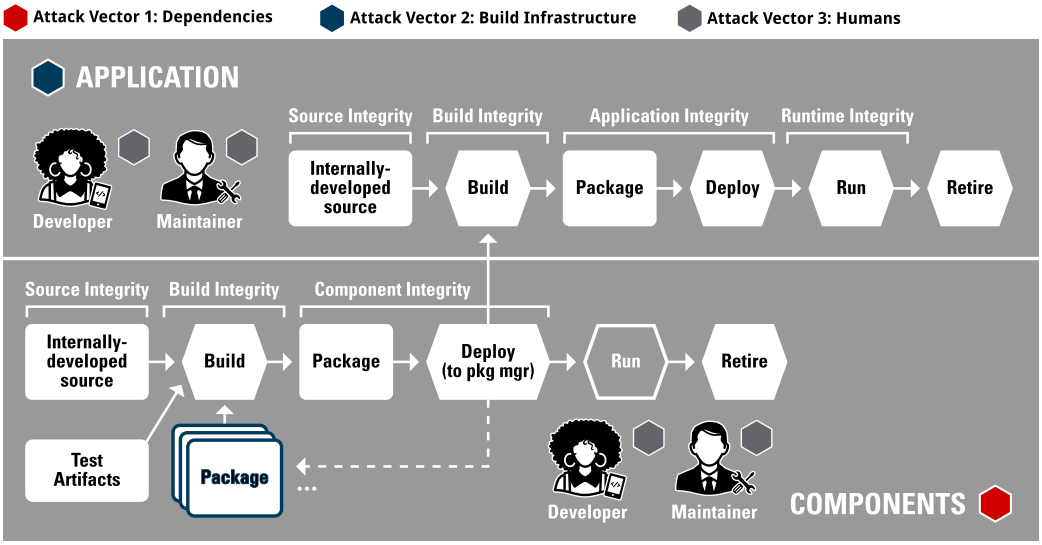


Fig. 1. Software supply chain attack vectors.

Second, attackers conduct software supply chain attacks by infiltrating the *build infrastructure* during the build and deployment processes. This attack vector appears in the upper "application" layer of Figure 1. Attackers launch attacks through the tooling and automation that turns the code of multiple software projects into the production application. For example, with SolarWinds [60, 204] the build process was compromised to inject malicious code into the end product that was signed and distributed to commercial companies, including a range of critical US government organizations and businesses. We discuss this attack vector in Section 4.

Finally, attackers conduct software supply chain attacks through targeted techniques aimed at the *humans* involved in software development, such as through social engineering. This attack vector appears in the lower "component" and the upper "application" layers of Figure 1 through the software developers and software maintainers of the components and the application. The humans involved in software development can be referred to as the software supply chain's "weak link." In the words of Ken Thompson in his Turing award speech [223], "To what extent should one trust a statement that a program is free of Trojan horses? Perhaps it is more important to trust the people who wrote the software." We discuss the humans attack vector in Section 5.

Plummeting trust in the software supply chain may decelerate digital innovation if software organizations reduce their use of open source and third-party artifacts to reduce risk. If so, these organizations would need to dedicate their own resources to implement functionality previously provided by open source and third parties, diverting resources from their proprietary innovation. Focused research is needed to develop fundamental principles, techniques, and tools to close the attack vectors in the software supply chain. This article provides an overview of research efforts to secure the software supply chain and proposes a future research agenda to close the software supply chain attack vectors and support the software industry.

The rest of this article is structured as follows. Section 2 provides an overview of the methodology used to obtain the knowledge shared in this article. Sections 3–5 provide an overview of challenges and research conducted to date for the three major attack vectors previously mentioned. Section 6 provides challenges and research results in other software supply chain directions. In Section 7, we conclude with our perspective on research directions in software supply chain security.

## 2 Perspectives

The authors of this article are members of or affiliated with the US National Science Foundation-funded Secure Software Supply Chain Center (S3C2; https://s3c2.org/). S3C2 is a large-scale, multi-institution research enterprise funded in 2022. The center members consist of researchers at North Carolina State University, Carnegie Mellon, and the University of Maryland, with a close collaborator at Paderborn University in Germany. Longer-term doctoral student visitors to S3C2 have also contributed to this article.

This article contains perspectives and knowledge obtained from intentional outreach with practitioners to understand their practical challenges (see Section 2.1) and from extensive research efforts (see Section 2.2).

### 2.1 Industry and Government Input on Challenges

Software supply chain security is a pressing practical problem facing the software industry. As a result, we knew our research needs to be informed by approaches used by and challenges faced by software practitioners. A central approach S3C2 uses to engage with the industry is conducting three Secure Software Supply Chain Summits annually. The goals of the Summits are to enable sharing between industry practitioners having practical experiences and challenges with software supply chain security; to help form new collaborations between industrial organizations and researchers; and to identify research opportunities.

Summit participants are recruited from companies and US government agencies, intentionally in diverse domains with various company maturity levels and sizes. To keep the event small enough that honest communication between participants can flow, each company or agency can have only one participant, except for the organization hosting the Summit, which may have two or three additional participants. The Summits are conducted under the Chatham House Rule, which states that all participants are free to use the information discussed, but the identity and affiliation of the speaker(s) and any other participant may be revealed. As such, none of the participating companies or agencies are identified in this article.

Between 2021 and 2024, we have conducted eight in-person and three virtual Secure Software Supply Chain Summits: eight with industry and three with US government. In these summits, 131 practitioners from 42 industrial organizations and 15 US government agencies participated. We have published summaries of the Summits [6, 49, 53, 227, 228, 261].

Additionally, S3C2 hosts a Secure Software Supply Chain Community Day annually, where practitioners and academics discuss challenges and research in a larger setting (50+ people). Finally, the researchers and students give presentations and attend events at industrial and government sites. The presentations serve as research dissemination and vehicles for understanding practical and research challenges in software supply chain security.

Throughout the article, we refer to the practical knowledge we got from these outreach activities as Practitioner Challenges.

### 2.2 Research

Most S3C2 resources are dedicated to conducting fundamental research to solve software supply chain security challenges. In this article, we share what we have contributed and what we have learned about other research efforts by conducting this research.

## 3 Attack Vector: Code Dependencies

Through the code dependencies attack vector, attackers inject vulnerabilities into upstream open source components such that they can be leveraged at scale to attack upstream projects. Modern

software products commonly have tens to hundreds of direct and transitive code dependencies, each of which can have vulnerabilities. Dependency relationships are within single language ecosystems and across different platforms, such as dependencies between Maven and PyPi packages to C/C++. To facilitate developers in understanding their direct and transitive dependencies, some visualization tools such as V-Achilles [98] and [21] have been developed. Remediating known vulnerabilities from code dependencies is labor-intensive [16, 22, 96, 113] and requires dedicated personnel. Malicious code dependencies have become increasingly common due to typo-squatting, dependency confusion, and project take-over attacks [117].

In this section, we provide practitioner challenges, research results, and open research challenges for the dependency attack vector.

### 3.1 Practitioner Challenges

In the following three sub-sections, we provide perspective on practitioner challenges related to code dependencies [6, 49, 53, 227, 228, 261].

*3.1.1 Dependency Choice.* Every dependency added to an application introduces both value and risk. Once incorporated into a project, teams rarely replace a dependency, so the initial choice of a dependency is important. Participants discussed various strategies for choosing dependencies and the lack of good metrics to aid their choice. The participants discussed the use of OpenSSF Scorecard [192] security health metrics. OpenSSF Scorecard assesses open source projects for security risks through 18 (in 2024) automated checks indicating that the development team used security practices, such as code review, branch protection, and fuzzing, or the project was actively maintained and vulnerabilities were fixed. Open source developers created Scorecard to help improve the health of critical projects on which the community depends. Only two participants indicated that they were using the Scorecard metrics in the component choice process. One participant said that they looked at Scorecard scores and measured whether or not packages were less likely to have vulnerabilities if their score was higher and found no relationship. This participant's observations were consistent with our studies of the relationship between Scorecard scores and vulnerability count [266]. Several other participants indicated they were evaluating how they might incorporate Scorecard into their component choice process.

The most common checks practitioners use before incorporating a component into their development are (1) whether the component is actively maintained; and (2) whether the component has unfixed known vulnerabilities.

The government participants discussed the realism of avoiding software dependencies developed in foreign countries. Participants discussed efforts to identify and manage when software comes from embargoed countries. However, these participants acknowledged that determined adversaries can find ways around those mechanisms.

In summary, practitioners are challenged to find a reliable and repeatable process for making good component choices.

*3.1.2 Dependency Update.* Practitioners commonly rely on different strategies and tools when updating vulnerable dependencies. Practitioners mentioned scanning for vulnerabilities by using **Software Composition Analysis (SCA)** tools. Multiple practitioners mentioned feeling overwhelmed by the number of vulnerabilities in direct and transitive dependencies identified by these tools. Tooling is available to automatically patch dependencies (e.g., Dependabot's automated pull requests). However, auto-patching without human intervention is often not an accepted practice. Concerns were expressed that the new versions of vulnerable dependencies may have breaking changes, so testing must be conducted. As a result, practitioners discussed intentionally delaying

updating their dependencies while passively "watching" that the dependency update did not cause problems for other organizations.

Even if all vulnerable dependencies were updated, a tool run the next day may identify new vulnerabilities. As a result, practitioners are challenged to catch up, and they are forced to triage and prioritize updating vulnerable dependencies. Practitioners desire information about whether the vulnerability in a component is reachable or exploitable and about the severity of the vulnerability in the product's context. Currently, no SCA tool provides sufficient information for prioritization. However, organizations need a process to stay up to date. A participant suggested that a good automated test suite that detects breaking changes can make a company more confident when updating a dependency.

*3.1.3 Malicious Commit Detection.* Actors in software supply chain attacks (e.g., SushiSwap) use commits to submit intentionally inject malicious changes to a source repository. Detecting and discerning these malicious commits is not straightforward. Attackers often use obfuscated code, steal authentication credentials, or use impersonation strategies to deceive and put malicious code changes through the system. Multiple practitioners believe a closer look at the committer's behavior might help discern malicious behavior. For instance, a committer's activity, reputation, and uncommon behavior (e.g., big vs. small changes, critical vs. ancillary fixes) can signal suspicious or malicious commits. **Machine Learning (ML)** could be applied to create appropriate tooling. Overall, the participants did not have satisfactory solutions for detecting malicious commits.

### 3.2 Research Results

Vulnerable dependencies expose applications to security breaches [7, 55, 64, 86]. Many studies have been done to find vulnerabilities in dependencies [8, 19, 130, 165, 166, 170, 196, 213]. On average, vulnerabilities remain undiscovered for approximately 3 years [43] but have different survival rates [269], highlighting the importance of proactive monitoring and detection of vulnerable dependencies.

This section summarizes research results related to the attack vector through components.

*3.2.1 Dependency Choice.* Providing meaningful quantitative measures for practitioners can help in choosing more secure dependencies. These measures include dependency popularity, contributor's reputation, and maintenance [83, 121, 219].

As discussed in Section 3.2.1, the OpenSSF Scorecard metrics can be used in dependency choice. Zahan et al. [264] analyzed the applicability of Scorecard by observing the resulting security scores within the npm and PyPI ecosystems. The results of the research indicated that 9 of 18 of Scorecard's automated metrics were reliably computable in both ecosystems and more work needed to be done to improve the automation of the other 9 metrics. Additionally, practitioners desire a demonstrated relationship between the adoption of security practices and improved security. In follow-up work, Zahan et al. [266] developed five supervised ML models for npm and PyPI packages using Scorecard's security scores as predictors and the number of externally reported vulnerabilities as dependent variables. Four security practices (Maintained, Code Review, Branch Protection, and Security Policy) were the most important practices influencing vulnerability count. However, the low $R^2$ (ranging from 9% to 12%) indicated that better metrics and models are needed to inform dependency choices.

*3.2.2 Dependency Update.* If not abandoned or deprecated, new dependencies' versions are released to incorporate new features, security patches, bug fixes, and refactor code [15, 26]. Yet adopting the up-to-date dependencies is not trivial for developers due to breaking changes, lack of

knowledge, and lack of motivation [45]. As such, dependencies are often outdated across ecosystems, with 32.2% of dependencies outdated in npm, 45.8% in PyPI, and 63.0% in Maven [210], respectively.

SCA tools provide information on vulnerabilities in direct or transitive dependency, regardless of whether the vulnerability is reachable or exploitable in the context of an application [77, 185]. The existence of a vulnerable software dependency does not mean that the corresponding vulnerable code is being used in a way that makes the application vulnerable [166, 271]. As such, practitioners may be asked to spend time on vulnerabilities that pose no risk, creating *notification fatigue*.

Approaches have been developed to assess the reachability [97, 101, 129, 171] and exploitability [97, 104, 258] of vulnerabilities in the software supply chain. Some SCA tools may provide this reachability/exploitability information. Imtiaz et al. [97] compared the vulnerability finding of several SCA tools. Similarly, Prana et al. [171] use commercial SCA tools to analyze the vulnerability reachability in Java, Python, and Ruby supply chain ecosystems. On the other hand, Liu et al. [129] use knowledge graph analysis to address the gap in vulnerability reachability analysis for ecosystems with dynamic dependency resolution like npm. However, more research is needed to produce tools that accurately provide exploitability and reachability information.

To prevent a project from automatically installing and using the latest release of a dependency, projects can *pin* a dependency to a specific version to reduce the risks of unintentionally incorporating untested or compromised dependency updates. Still, balancing the predictability of pinning versus the agility of automatically including dependency updates is difficult for developers [46]. Work has examined how dependencies are updated, with associated code smells, across ecosystems [28, 46, 99].

Research has worked on quantifying the amount of time a dependency is out-of-date. Dependency freshness [39] was one of the first measures, quantifying the difference between the used and the desired dependency version. Additionally, work has predominantly investigated technical lag that measures the delay between the used version of the dependency and the latest version of the dependency in terms of the time difference (time lag) or version difference (version lag) [42, 210, 268]. To understand how dependencies evolve, Rahman et al. [178] introduced Mean-Time-To-Update and Mean-Time-To-Remediate.

*3.2.3 Malicious Commit Detection.* Beyond accidentally introduced vulnerabilities in dependencies, a significant concern in software supply chain security is the ability of threat actors to intentionally commit new malicious packages [193] or updates to existing packages by gaining access to that package [74]. The xz utils attack [64, 84] is an example involving malicious commits. Past attacks have been widely discussed, especially malicious package updates [59, 69, 191], undermining the trust in software packages after initial adoption. This sub-section summarizes research conducted to detect malicious packages in the software supply chain.

Research and techniques to detect malicious packages can be classified as rule-based and heuristic [48, 69, 74, 94, 118, 267], differential analysis [14, 66, 155, 191, 237, 239], ML [94, 119, 154, 193] and **Large Language Model (LLMs)** [201, 260, 263, 272] approaches. Rule-based approaches often rely on predefined rules about package metadata (e.g., package name), suspicious imports, and method calls. Many studies [48, 69, 74, 267] have focused on detecting static features, such as libraries that access the network, file system access, features in package metadata, dynamic analysis, or language-independent features like commit logs and repository metadata to flag packages as suspicious using heuristic rules. Huang et al. [94] constructed a behavior knowledge base by analyzing API call sequences and using static and dynamic analysis to classify suspicious behaviors, focusing on entry points and obfuscated code. Differential analysis involves a comparison of a previous version with a target version to analyze differences and identify malicious behavior.

Differential analysis research [66, 155, 191, 237, 239] leverages artifacts, file hashes, and "phantom lines" to show differences between versions of packages and source code repositories.

Studies [154, 193] have evaluated various supervised ML models to detect malware. Research [201, 260, 263, 272] has also leveraged LLM to detect malicious packages. Zahan et al. presented SecurityAI [263], a complete malicious code review workflow to detect malicious code using **Generative Pre-Training Transformer (GPT)** models, and achieved high accuracy in model performance. Zhang et al. [272] used the BERT model to understand the semantics of malicious behavior. Yu et al. [260] used LLM to translate malicious functions from other languages (e.g., C/C++, Python, and Go) into JavaScript. Singla et al. [201] used GPT and Bard models to characterize software supply chain failure reports/blogs into the type of compromise, intent, nature, and impact.

Additionally, multiple research studies [220, 239] exist on detecting typosquat attacks in which the attacker leverages common misspellings a user may incorrectly type or may not perceive in a URL or package name. Vu et al. [238] studied the PyPI ecosystem to evaluate the effectiveness of current malware detection tools, including Bandit4Mal [236], and OSSGadget's OSS Detect Backdoor [137], using a benchmark dataset. Their analysis revealed consistently high false-positive rates, over 15%, making automated detection tools impractical for repository administrators, who require near-zero false positives. Interviews with PyPI administrators and security researchers highlighted a socio-technical system in which external researchers play a key role in detecting and reporting malware. The authors recommend strengthening collaboration between administrators and researchers, aligning their incentives, and improving coordination and tooling to enhance ecosystem security. Ladisa et al. [120] studied attackers exploiting package managers to achieve Arbitrary Code Execution in open source software supply chains. The study analyzes seven ecosystems, including npm and PyPI, identifying three install-time and four runtime techniques that attackers use to execute malicious code.

*3.2.4 Dependency Decommission.* A significant body of work has researched how to sustain software, focused on how to keep dependencies alive and recruit new contributors [209, 233]. Still, the long-term viability of dependencies being updated is unknown and cannot be guaranteed. Dealing with deprecated functions or API calls during development is not uncommon. Yet, a more significant concern is not only when feature development stops but when the dependency is abandoned, turning software reuse from a resource-empowering rapid innovation tool into a liability. In practice, even popular dependencies become abandoned. For example, in the npm ecosystem, out of the most popular dependencies with over 10,000 monthly downloads, 3,000 dependencies are abandoned [139]. The response time to fix any security issues in abandoned packages is uncertain, and leaving end-users potentially vulnerable due to unawareness of the security issues as there are no active maintainers to fix the issues [267]. Therefore, dealing with the end-of-life of a software dependency is necessary.

Research has worked on understanding and detecting abandoned dependencies [138, 143]. As many dependencies are open source projects that rely on volunteer contributions, the abandonment of open source projects is of particular research interest. Of note, Miller et al. [139] interviewed practitioners to understand strategies for navigating dependency abandonment. Strategies found included forking the dependency, somehow encouraging others to maintain it, or moving to an alternative if it exists, often at substantial cost. More research and development should focus on navigating inevitable situations where projects are no longer maintained, whether through deliberate sunsetting strategies or community-based efforts to help developers cope with abandoned dependencies.

*3.2.5 Residual and Orphaned Vulnerabilities.* Vulnerability fixes may not be propagated to previous versions, which results in residual vulnerabilities. Yan et al. [254] assessed the attack surface caused by residual vulnerabilities. Additionally, orphan vulnerabilities happen through

copying dependencies. Reid et al. [180] conducted a large-scale empirical study of orphaned vulnerabilities in the software supply chain.

*3.2.6 Vulnerability Patch Information.* Approaches and tools for vulnerability remediation benefit from patch information. The most common fix information is a code commit that fixes the vulnerability, often called the **"Vulnerability Fixing Commit" (VFC)** or simply the "patch link." As will be discussed in Section 3.2.8, the data fields in databases, such as the **National Vulnerability Database (NVD)** [153], are often incomplete, with many vulnerabilities without a patch or **Common Vulnerabilities and Exposures identifier (CVE ID)** [112, 135]. For instance, 63% of security advisories in the GitHub Security Advisory (GHSA) Database for open source software lack patch links [50]. Similarly, 71% of vulnerabilities in NVD and 46% in Snyk also lacked patch links [190], making it difficult to trace and apply necessary fixes.

To this end, research efforts have been conducted to provide patch link/VFC information [50, 88, 123, 146, 147, 187, 190, 218, 280, 281]. A VFC can be enriched following reference links in the advisories [251]. However, similar techniques suffer from poorly documented security commit messages and incomplete security advisories limiting their effectiveness. Dunlap et al. [50] have created a tool called VFCFinder, which uses machine learning to backfill VFCs for a given security advisory. VFCFinder provides a list of the Top 5 potential VFCs. The VFC is in that list over 96% of the time. VFCFinder's top choice is accurate 80% of the time, and if there are fewer than 15 commits between releases, over 90% of the time.

*3.2.7 Vulnerability Remediation.* Several works have been introduced to remediate vulnerable dependencies by avoiding vulnerable dependency inclusion, such as using `npm audit fix`, Dependabot [70], or other developed tools [129, 242, 273, 274]. To avoid resolving vulnerable versions of dependencies, Mohayeji et al. [140] found that developers often delegated the task of upgrading vulnerable dependencies to a secure version to Dependabot and merged them within several days. Otherwise, the developers manually address the vulnerability, which takes longer to complete. Instead of manually addressing the vulnerability, Liu et al. [129] proposed DTReme, which uses their knowledge graph-based dependency resolver to explore all possible solutions for a single dependency. With a similar idea, Ranger [273] was proposed to restore secure version ranges of dependencies. On the other hand, CORAL [274] was proposed to not only find the secure dependencies but also make sure that they are compatible with the current code base. Similarly, Plumber [242] was proposed to propagate vulnerability fixes throughout the dependency tree (i.e., finding secure dependencies) and use these secure dependency paths to remediate the vulnerabilities in the npm ecosystem. Another approach is to use active learning and regeneration to "fix" vulnerable dependencies [234].

*3.2.8 Databases for Research and Tool Development.* Vulnerability information is important for dependency risk evaluation, as it helps identify security risks in third-party dependencies while guiding tool developers to enhance detection capabilities. Tools like Dependabot [70] and Renovatebot [184] depend on vulnerability databases to notify developers of known issues in dependencies. In this sub-section, we discuss existing efforts to construct vulnerability databases.

*Vulnerability Databases.* Several platforms exist for vulnerability databases to track vulnerabilities in dependencies. The US government maintains the NVD.[1] The Open Source Vulnerability (OSV) [76] aggregator compiles vulnerabilities from diverse sources, such as GHSA [71], PyPI Advisory Database [173], Go Vulnerability Database [240], Rust Advisory Database [186], OSSFuzz (mostly C/C++) [75], and Global Security Database [9]. The aggregators provide both human- and machine-readable formats for easier integration into security tooling. Similarly, platforms like Snyk [203] curate their vulnerability reports by supplementing publicly available

[1] https://nvd.nist.gov/

data with their findings. In addition, specialized databases, like SafetyDB [174], focus on specific ecosystems, such as Python's PyPI, curating vulnerability data that include CVE IDs and patch links where available.

From these databases, several vulnerability datasets are crafted for ML-based vulnerability detection, such as BigVul [57] and Devign [282]. Some datasets use CVE-based labeling, while others are synthetically/semi-synthetically labeled. However, Croft et al. [40] found that these datasets have some issues, such as accuracy and uniqueness.

Despite the advances and availability of these databases, the fragmented nature of vulnerability databases still poses challenges. Not only fragmented, Wu et al. [247, 248] also found that these databases are often inconsistent with each other, in terms of affected library names, ecosystem, and versions. They then proposed a method to identify affected libraries and versions accurately.

Other than inconsistencies, public vulnerability datasets/databases also miss some "silent" vulnerabilities. In academia, a line of research seeks to discover "silent vulnerability fixes" [146, 212] to find missing vulnerabilities without CVE IDs. For example, Dunlap et al. [52] published an approach called Differential Alert Analysis that uses commercial-of-the-shelf Static Application Security Testing tools to identify when a developer commits code that fixes a vulnerability. With limited resources, they identified 111 silent security fixes in the npm, Go, PyPI, and Maven ecosystems. The authors' subsequent efforts [50] have shown how LLMs, such as CodeBERT, can further discover silent vulnerability fixes. In addition to missing some vulnerabilities, these public vulnerability datasets also lack fix information. We discuss some efforts to find missing fixes in Section 3.2.6.

*Malicious Package Databases.* Package managers remove malicious packages from the registry and replace the original metadata with dummy metadata so malicious packages do not create further risk. As a result, researchers cannot extract, research, and study meaningful information once a package is removed from the repositories. As a result, only a few malicious packages are available to study in deprecated mirrors, internet archives, and public research repositories.

Several studies have contributed to constructing malicious package datasets within the npm and PyPI ecosystems. The Backstabber's Knife Collection dataset [156] includes 2,829 malicious packages, covering multiple ecosystems such as npm, PyPI, and RubyGems, identified through manual analysis and public advisories. Guo et al. [81] created a dataset of 2,416 malicious PyPI packages using a semi-automated method to obtain code from open databases, IT technology websites, code hosting websites, and official software repositories. Datadog [115] released a dataset of 1,397 malicious PyPI packages collected using the GuardDog tool [41], obtained through manual triaging. The MALOSS dataset [48] includes 852 npm and PyPI packages, identified through metadata and program analysis. Although prior works [48, 81, 115, 156] constructed malicious package datasets, the studies did not incorporate neutral package data into their labeled datasets. When a package has no discovered malicious code, a package is neutral [262]. Labeled benchmark datasets enable tool developers and researchers to differentiate between malicious and non-malicious packages in prediction models, thereby reducing false positives. In 2024, Zahan et al. [262] presented MalwareBench, a labeled dataset of 20,792 malicious and neutral packages of npm and PyPI ecosystems. MalwareBench incorporates data from existing malicious package datasets [48, 81, 115, 156], as well as the Socket [4] internal malicious dataset and including popular, randomly selected and newly released npm and PyPI neutral packages. The ground-truth labels for the benchmark were established using a hybrid approach: researchers manually reviewed and confirmed all malicious cases flagged by automated tools, while neutral cases were initially identified through automation and further validated via manual analysis of randomly selected samples [263].

### 3.3 Open Research Challenges

This section discusses the challenges we have encountered while conducting research in code dependencies.

*3.3.1 Dependency Choice.* Selecting and updating dependencies is not a trivial matter. Dependency updates occasionally break functionality (e.g., by changing APIs) or introduce new bugs, while poorly maintained or abandoned dependencies may contain vulnerabilities or malicious code. Developers face difficulties manually assessing dependencies with their transitive dependencies for security risks. Developers could use security-relevant information (e.g., security-sensitive API calls [177]) or tools such as Scorecard to assess the security posture of a dependency. Such metrics can serve as a criterion for dependency selection. However, prior research also showed such metrics need improvement and often rely on previous, incomplete CVE information. For example, prior studies [264, 266] have highlighted that measurement tools such as Scorecard [192] require improvements and suffer from a lack of consensus on ecosystem-wide standards among ecosystem managers, tool developers, and users. Future research should focus on developing, improving, and leveraging dependency measures for practitioners.

*3.3.2 Databases for Research and Tool Development.* Vulnerability and vulnerable package data are needed to enable research and tool development.

*Existing Vulnerability Data.* Though efforts have been made to improve the completeness of current public vulnerability datasets, the databases still lack information. More research is needed to integrate currently available tools, such as [52, 212], to add more "silent vulnerabilities" to the public vulnerability databases. Automatically generating the CVE entries after vulnerable commits are found is also a research gap. Moreover, varying data completeness and formats across different platforms reduce the effectiveness of dependency scanning tools. Research is needed to improve these databases to have a convention on what data has to be available and which format needs to be used to improve the effectiveness of supply chain analyses.

*Existing Malicious Data.* We have limited reliable information on malicious dependencies. Package ecosystems tend to delete malicious dependencies once reported, and researchers have limited access. Many dependencies are taken down within minutes of their release. Our initial explorations of dependencies removed as malicious from npm reveal that many of them are copies of the same dependency under different names, and almost all of them are new dependencies rather than updates to existing ones. Prior studies [81, 115, 262, 263] have discussed the lack of diversity in malicious dependency datasets, limiting practitioners' ability to assess the performance detection tool for complicated supply chain attacks.

Many datasets and software supply chain security reports do not distinguish different kinds of attacks. For example, distinguishing typosquatting from malicious updates to popular dependencies from dependencies taken down simply as spam (a rapidly increasing category). More detailed reporting and automated classification are needed to provide a more reliable understanding of security trends in software supply chains beyond the flashy numbers counting all attacks common in today's discourse. Encouraging operators of package managers to report details and share removed artifacts with researchers could provide a much better understanding.

*3.3.3 A Gap between Theory and Practice in Tool Integration.* Many researchers advertise their detection techniques, including automation, and achieve high accuracy in detecting malicious dependencies. However, such tools rarely acknowledge the unique requirements of their setting in the **Software Development Life Cycle (SDLC)** [238]. In most cases, the important criterion is that a check must provide a "useful signal" and a low false positive rate. Future research needs to

keep the SDLC in mind. Additionally, researchers need to consider developers' and administrators' perspectives while proposing academic or industrial tools for scalability.

*3.3.4 Cross-Platform Research.* As we discussed in Section 3.2, there are a number of problems to solve concerning dependency and vulnerability detection, tracking, and response. One of the common challenges is the lack of a standardized format for data interchange across multiple vulnerability and ecosystems API databases. Researchers and clients attempting to collect information from multiple databases must handle each database or API completely differently. Additionally, if practitioners want to exchange information between different databases, they need to develop specific parsers for each unique format. Such a lack of standardization and interoperability complicates the tracking of dependencies and hinders collaboration between different vulnerability databases and ecosystem efforts. The OSV [76] vulnerability database is an effort in that direction. However, other databases must make the entries available in OSV format to enable interoperability. Ecosystem package managers need to work together to provide the same format of metadata for researchers or clients to use efficiently.

## 4 Attack Vector: Build Infrastructure

The process and tooling enabling deployment of code, known as the build infrastructure, is another notable attack vector within the software supply chain. The build infrastructure attack vector was exploited in December 2020 during the SolarWinds software supply chain attack, injecting malicious code into the end product. Until recently, build systems have seen relatively little research attention compared to code dependencies.

### 4.1 Practitioner Challenges

Build platforms and **Continuous Integration and Continuous Deployment (CI/CD)** tools have the potential to enhance software build integrity by establishing documented and consistent build environments, isolating build processes, and generating verifiable provenance. Most practitioners feel comfortable in securing the deployment process and are more worried about the build process. Overall, companies seek guidance on secure build and deployment from the **Supply-Chain Levels for Software Artifacts (SLSA)** framework [161]. Most practitioners believe reproducible builds may not be a practical means for check whether a build has been tampered with as there still many challenges and concerns with their generation. As of now, only 20% of builds match bit-to-bit [6, 49, 53, 227, 228, 261].

### 4.2 Research Results

In this section, we summarize research findings for the build infrastructure attack vector.

*4.2.1 Transparency.* In software supply chain security, transparency in the build infrastructure is essential for ensuring traceability and accountability. Transparency allows monitoring and linking each phase of the build process with specific contributors. Hence, we can detect unauthorized and malicious commits, aiding in dependency integrity. With a verified and visible build process pipeline, Security frameworks and technologies such as SLSA and the use of a SBOM emphasize transparency in build systems to reduce the risks associated with software supply chain attacks.

*SLSA.* The SLSA framework provides transparency in build systems through the use of transparent logs, signed artifacts, and auditable processes [160]. SLSA has four levels; as the level increases, it provides greater controls and transparency in the build and deployment processes. Provenance generation is a key aspect of SLSA that includes detailed information about the source code, reproducible builds, build systems, dependencies, and environments. The framework also incorporates strongly authenticated and access-controlled version control, dependency tracking, and the use

of cryptographic attestations to enhance visibility. Additionally, SLSA's automation and auditable records help to reduce human error and improve incident response. Furthermore, integration with SBOM practices provides a comprehensive view of all components and dependencies within a software artifact [202]. However, studies have identified challenges in incorporating SLSA in software projects [217]. By addressing these challenges, SLSA can enhance transparency in the software supply chain to improve overall transparency and trustworthiness.

*4.2.2 Integrity.* Being able to see everything that goes on during a build process is not enough to protect against attacks against the final build artifact. If attackers are able to inject code into the final build artifact through unconventional means, attackers' actions may not be captured with transparency tools alone. Integrity protections are needed at every step of the software supply chain building process to verify included artifacts and developers' identity. In this section, we discuss research conducted to detect whether a piece of software has been tampered with.

*In-Toto.* Software development involves multiple stages, from source code submission to packaging for distribution, each managed by independent actors in the software supply chain. A compromise at any step, such as code or CI/CD testing, can allow an attacker to inject malicious code, potentially harming end users. Torres-Arias et al. [225] designed in-toto to cryptographically ensure the integrity of the software supply chain that exists between the code in the developers' source code repository and the code that is shipped in an end product. In-toto is designed to achieve five security goals: software supply chain layout integrity, artifact flow integrity, step authentication, implementation transparency, and graceful degradation of security properties. The goals are accomplished by following a developer-made layout and gathering information about the inputs and outputs of each defined build step. Each step requires cryptographic signatures made by trusted developers who are assigned specific roles. The layouts and roles are created at the project conception but can be changed throughout the development process only with approval from the project owner. In-toto has achieved widespread adoption, notably as the tool provides the integrity guarantees used for SLSA provenance.

*TUF.* While in-toto provides end-to-end verification of a software supply chain for transforming software from source repository to the end package, it does not address how to securely distribute, revoke, and replace the public keys used to verify the in-toto layout [225]. Samuel et al. [188] designed and implemented "**The Update Framework (TUF)**," a software update framework to minimize the impact of key compromise. TUF metadata provides information that clients can use to make updated decisions. TUF by itself does not update the software; rather, it secures the system that automatically identifies and downloads updates to the software. TUF leverages mechanisms such as roles, their signatures (public key infrastructure), threshold number of signatures, file hashes, and file size. All the information is provided as metadata files, which are version-numbered and have an expiration date. Each role requires a threshold of $m$ out of $n$ signatures for acceptance. Thus, a single key compromise does not impact the whole system's security. The Linux Foundation hosts TUF as part of the **Cloud Native Computing Foundation (CNCF)** [37], and is used in production by a growing number of companies like such as Datadog [54], IBM [111], OpenSSF [163], and the Python Packaging Index [211].

*Sigstore.* Managing keys to sign software is hard. Newman et al. propose Sigstore [145], a system to provide baseline artifact signing capabilities that minimize the adoption barrier for developers by not requiring them to manage long-term signing keys. Sigstore enables user authentication using artifact and identity logs through three-phase process: Trust Setup, Signing Flow, and Verification Flow. Trust Setup is achieved by TUF as the root-of-trust. TUF manages key material, rather than artifacts, and distributes root certificates for Fulcio (Certificate Authority and Identity Log) [198] and Rekor (Artifact Log) [200]. In the Signing Flow, Sigstore integrates with three systems: an OIDC [199] provider for identity verification, Fulcio for issuing *short-lived* certificates, and Rekor

for logging artifact signatures to minimize key management for signers. The Verification Flow leverages TUF to ensure that root keys are fresh and valid, then checks that the identity was correctly verified by the identity log and confirms that the signature matches the artifact, all within the valid certificate time window. Sigstore has quickly become a critical piece of Internet infrastructure with more than 2.2M signatures over critical software such as Kubernetes and Distroless [145].

*gittuf.* *gittuf* [256] is a sandbox project at the Open Source Security Foundation (OpenSSF), providing a security layer for Git using concepts introduced by TUF. Git does not manage which developer keys should be trusted for a commit or tag signatures in a repository. gittuf allows for the association of trusted keys using Sigstore identities for developers who use gitsign, so developers can sign their Git commits.

*4.2.3 Trust.* Trust is paramount for the successful and secure functioning of software supply chains.

*Developer Trust.* Trust in developers is critical in the open source ecosystem, where interconnected communities rely on one another to maintain software. Trust relationships are inherently human, often placing trust in individuals or small teams responsible for key components of foundational software infrastructure. Research efforts have focused on analyzing these trust relationships in open source systems [24], offering a process to decompose trust, identify key relationships, and define behavior-based indicators.

Modern software installation tools frequently pull dependencies from multiple repositories, increasing the risk of repository compromise and introducing attacks like dependency confusion and repository fallback. Moore et al. [142] proposed a novel defensive strategy called articulated trust and built Artemis, a security framework that enhances trust in software installation tools by allowing them to specify trusted developers and repositories on a per-dependency basis. The framework incorporates key techniques like per-dependency repository prioritization, multi-role delegation, multiple-repository consensus, and key pinning. The framework aids in mitigating single points of failure and enhances the security of multi-repository systems by introducing flexible, user-defined trust models.

*Software Trust (Reproducible Builds).* The SolarWinds attack in 2020 highlighted a large amount of trust placed in build systems, with attackers injecting malicious logic into the product binary signed with Solarwinds' official code signing keys. Reproducible builds provide a strong foundation to build defenses for arbitrary attacks against build systems by ensuring that given the same source code, build environment, and build instructions, bitwise-identical artifacts are created.

However, most software builds are not reproducible. Previous research has studied how to automatically localize code and commands producing unreproducible builds [181, 182] and automated approaches for patching [183]. Still, most of the software industry considers reproducible builds a long-term goal [27].

To understand the reasons behind it, Fourne et al. [63] conducted 24 semi-structured expert interviews with participants from the Reproducible-Builds.org project. The research found that commonly encountered obstacles to reproducible builds include build directory name inclusion and cryptographic signatures on the technical side, as well as the need for patience and good social communication on the interaction side. Many interviewees mentioned positive interactions with upstream projects and other developers, although some specifically noted that upstream communication required patience. As for helpful factors, most mentioned were being self-effective, (being determined, possessing the skill-set to progress reproducible builds), and having good communication with other developers. For transitive dependency problems, concrete technical documentation could be achieved by the pervasive use of SBOMs to indicate all software included in building an artifact, so the transitive dependencies could be traced over a dependency graph. Some

interviewees suggested that the overall awareness and buy-in for reproducible builds were lacking and that even with the increase in the prevalence of software supply chain attacks, reproducible builds are not yet widespread.

*4.2.4 CI/CD Vulnerabilities.* Modern software development has increased in complexity and several parts of it, such as building, testing and deploying, require automation. As such, CI/CD pipelines have become essential to help in automating the development process. Yet, these pipelines introduce new code and dependencies that may introduce security bugs, posing new risks to software projects [126]. CI/CDs can be plagued by similar security issues found in other domains, such as caching attacks, which can leak critical data to attackers and allow them to take over pipelines [78]. Currently, the trend of research in this area has focused on developers' perceptions and secure practices [29, 44, 58]. The ideas here revolve around adding security tools into the pipeline throughout the development process and not just at the end before the product gets released.

Insecure practices not only apply to developers creating the pipelines but also CI/CD and code hosting platforms that enable the creation of them. Prior works have looked into the insecure usage of secret tokens [79, 136]. These tokens have a variety of uses such as controlling multiple processes throughout the pipeline (task assignment, task execution, artifact uploading, etc.) and authenticating to third-party applications. Gu et al. [79] performed a systematic study to uncover the novel threats of task hijacking, repository privilege escalation, task result hijacking, and artifact hijacking across seven major CI platforms spanning three major code hosting platforms. These attacks were made feasible because all code hosting platforms are susceptible to secret leakage due to improper configurations (resource sharing, over-privileged, improper validity periods).

Even though these studies focus on trying to protect the code as it flows through the pipeline, they tend to not cover (or handle) the unique problems that arise from specialized pipelines such as GitHub Actions.

Previous research [18, 109] has tried to find vulnerabilities in GitHub Action pipelines, also known as GitHub Workflows, using pattern-matching. The current focus has been on GitHub Actions, a CI/CD integrated directly into GitHub, as it has become widely used due to GitHub being the most popular platform to host code [232]. However, there are significant challenges in automatically analyzing these pipelines for security problems. First, GitHub Workflows consist of jobs, each containing a sequence of steps, leading to a non-linear execution model. In addition, a job can depend on one or more other jobs, creating a complex interwoven build system.

Automatically analyzing this new execution model for code injection vulnerabilities requires new analysis techniques that consider the non-linear execution semantics of workflows. A workflow can also reference third-party Actions and can participate in sensitive operations, such as access to untrusted input and passing data to dangerous sinks. GitHub also supports three types of Actions: JavaScript, Composite (combine multiple workflow steps within one action), and Docker. The flexibility provided poses significant challenges for analyzing GitHub Actions.

Muralee et al. [144] studied if code injection vulnerabilities are prevalent in GitHub Actions by building a static taint analysis system called Argus. Argus pinpoints potential threats by tracking the flow of untrusted data from sources to sensitive sinks using a Workflow Intermediate Representation to address the non-linear execution semantics of workflows. Argus was run on nearly 2.8 million Workflows (1 million repositories) that utilized over 30,000 Actions. It detected high and medium impact code injection vulnerabilities in 4,307 workflows, posing a significant risk of compromising the respective repositories. Additionally, 80 vulnerable Actions were identified, which rendered any Workflow that uses them vulnerable. This led others to start looking into actions as if they were dependencies of the build process itself, similar to dependencies of the

source code. As such, dependency problems (such as using vulnerable dependent packages) were found to also affect actions [159].

### 4.3 Open Research Challenges

While some work has been done focusing on build infrastructure, it currently lags behind other areas of the software supply chain. Here we discuss some of the current research challenges that remain open for future work.

*Transparency.* To provide transparency, provenance generation is essential but implementing provenance and verifying integrity is considered as a complex process. Organizations seek a solution that enables stakeholders to easily generate provenance, store it securely, and provide verification. At present, SLSA provides a default open source solution for GitHub but many organizations work in varied environments, utilizing several Source Code Management and CI/CD systems. To be effective, provenance generation functionalities need to be accordant with a wide range of CI/CD platforms beyond just GitHub [67, 217].

For provenance generation to be effective, provenance generation must be compatible with a wide range of various CI/CD platforms, extending beyond just GitHub Moreover, since SLSA depends on cryptographic attestations for verification, additional research is necessary to ensure the secure management of cryptographic keys to prevent any misuse.

Integrating SBOM generation into the build process can be a turning point for adopting SBOM. Different software ecosystems use various strategies for dependency resolution, complicating the work of SBOM generation tools. Including SBOM generation inside of package managers provides better SBOMs that can be used for conducting better vulnerability analyses [17]. However, multiple languages can contribute to creating a single project, and then a cross-ecosystem approach is necessary for SBOM generation.

*Integrity.* As described in Section 4.2.2 great advances have been made in the infrastructure needed to support code signing and artifact provenance. However, code signing is still in the early stages of adoption by practitioners and many packages and components are not signed. Increasing adoption will depend on both technical means, e.g. removing friction in the concrete aspects of the code signing process, and social acceptance, where it becomes a habit for teams.

*Trust.* Even if the build code is free of vulnerabilities, build servers could be compromised. Reproducible Builds provide a strong primitive for protecting against such subversion. Unfortunately, the code that builds software has many sources of non-determinism that prevent reproducible builds from being widespread. While significant effort has been made to make core dependencies reproducible, more is needed to take Reproducible Builds to the last mile.

*CI/CD Vulnerabilities.* Software threat models need to include the code that builds the code, as vulnerabilities can also be found in the CI/CD codebase. However, assessing risk in build code can be challenging. For example, GitHub Actions can execute Actions in containers, introducing unique vulnerabilities, including privilege escalation and container escape scenarios. Current research has been limited to GitHub Actions, and there is a need to expand tools to other CI/CD ecosystems and cross-ecosystem analyses.

## 5 Attack Vector: Humans

In cybersecurity, attackers do not solely rely on exploiting technical flaws; they increasingly target the humans behind the technology. In today's interconnected world, the responsibility for security no longer rests solely with dedicated defenders like administrators and security specialists. Ordinary users now bear a significant share in protecting systems. While defenders must be perfect every time, attackers only need to get lucky once—finding a single vulnerability or exploiting a minor

mistake to breach the system. Attackers are aware of this imbalance and actively exploit it, making humans a critical and vulnerable attack vector in the software supply chain.

### 5.1 Practitioner Challenges

On the practitioners' side, the 2021 Executive Orders on *"America's Supply Chain"* [222] and *"Improving the Nation's Cybersecurity"* [5] significantly impacted software supply chain security practices, driving the adoption of new approaches, metrics, frameworks such as NIST's Secure **Software Development Framework (SSDF)** [150], OpenSSF's SLSA [161], Microsoft's **Software Supply Chain Consumption Framework (S2C2F)** [162], OWASP's SCVS [164], CNCF Foundations' SSCP [36], and combined mapping **Proactive Software Supply Chain Risk Management (P-SSCRM)** [246]. These frameworks rely on widespread industry adoption, standardized metrics, and active information sharing, such as SBOMs, to be effective. Effective supply chain security depends on collective efforts and industry-wide adoption. Without broad implementation of frameworks and practices, progress is limited. Major industry initiatives like those under the Linux Foundation and OpenSSF highlight the importance of this collaboration.

Summit attendees discussed adapting their cultures post-incidents like SolarWinds, emphasizing the role of "scenario owners" and regular meetings [261]. Encouraging trusted sources for external code, like GitHub, and reducing *ad hoc* fixes helps maintain integrity, although deadline pressures can still lead to risky decisions. Companies have adopted a "guardrails, not gates" approach to allow developers to move quickly within secure boundaries [261]. Tools like the "paved path" help maintain secure workflows with minimal disruption. Automation of vulnerability fixes reduces the burden on developers, enabling scalable security without relying on individual adherence to secure coding practices. Building a strong security culture requires company-wide commitment, with security seen as a shared responsibility.

### 5.2 Research Results

*5.2.1 Supporting Developers.* Implementing software supply chain security is a complex endeavor, involving many stakeholders and requiring developers to navigate challenges like establishing trust in third-party components and ensuring that external code is vulnerability-free and current. The reliance on open source components amplifies these challenges, as companies inherit unique security risks by incorporating code from potentially unvetted contributors. In the recent 2024 Tidelift "State of the Open Source Maintainer Report" [224], only 14% of open source maintainers reported they implemented formal processes or standards to prioritize the order in which pull requests and issues are addressed for most or all projects they maintain. Research into the area of open source components in 25 in-depth interviews with developers, architects, and engineers [244] finds that many companies have basic policies or best practices for including external code, but developers express a need for more resources, such as dedicated teams or tools, to better audit and secure these components. The authors underline the importance of companies contributing back to the open source ecosystem as a way to ensure mutual security for everyone rather than treating it solely as a free supply chain.

The complexities of software supply chain security mirror broader challenges in the development of security and privacy software, where, despite positive intentions, unintended consequences can be introduced. In a study involving 14 expert interviews with security and privacy software creators [179], researchers explored how these creators address the unintended impacts of their tools. They found that such consequences are often overlooked or handled in an ad hoc manner, usually based on user feedback, which places the burden on users to identify and report issues. To better address these challenges, the researchers recommend that developers proactively anticipate

potential negative impacts by increasing awareness, promoting accountability within organizations, and utilizing systematic toolkits to foresee and mitigate these risks.

*5.2.2 Open Source Ecosystem.* Open source software underpins much of modern technology, comprising the majority of the software supply chain in both consumer and critical infrastructure: a 2023 Synopsys report found that 96% of 1,700 codebases included open source components, accounting for 76% of the code [214]. Similarly, a 2021 Sonatype report noted that over 37 million open source components are used by 27 million developers across major ecosystems like Java, JavaScript, Python, and .NET [205]. However, decentralized development and open collaboration in these projects introduce challenges, such as code from unknown contributors, limited resources for reviews, and difficulties in onboarding new contributors. The recent 2024 Tidelift "State of the Open Source Maintainer Report" [224] reported that 60% of maintainers are not paid for their work and are spending almost three times more time (11%) on security work than they reported in 2021 (4%).

In a study involving 27 in-depth, semi-structured interviews with project owners, maintainers, and contributors [245], researchers examined the behind-the-scenes processes of open source projects, including their guidance, policies, incident handling, and challenges encountered. The findings include a wide variation in security measures and motivations across projects: Larger projects develop more structured security and trust processes as they scale, while smaller projects handle security on an *ad hoc* basis due to limited resources and frequent turnover. The research underscores the need for tailored support for open source projects, especially smaller ones with fewer contributors and resources. On the human side of open source, broader support is needed to prevent issues like project abandonment or hostile takeovers, exemplified by the xz-utils case.

Further research examined the challenges of secure code secret management in collaborative development environments like GitHub and GitLab. Secrets, such as API keys or passwords, are unintentionally leaked to public repositories, leading to serious security risks. In a mixed-methods study involving a survey of 109 developers and 14 in-depth interviews [110], 30.3% of participants had encountered secret leaks in the past.

*5.2.3 Academia.* Researchers in software supply chain security encounter complex ethical dilemmas: disclosing vulnerabilities in no longer maintained software can either help mitigate risks or unintentionally expose users still relying on it, studying data obtained from leaks raises issues around privacy and consent, and conducting security research with human participants requires reflections about informed consent, privacy, and the potential harm to participants. Recent work has significantly shaped how the computer security field approaches ethical research practices [108], influencing guidelines like those used by USENIX Security for evaluating the ethics of submitted papers [231]. The work explores the intersection of moral philosophy and computer security ethics through the creation and analysis of security-themed moral dilemmas inspired by trolley problems. Rather than defining right or wrong, the researchers' goal is to encourage meaningful discussions among security researchers about the ethical dimensions of their decisions.

### 5.3 Open Research Challenges

Going forward, the widespread adoption of software supply chain security approaches such as frameworks, standardized metrics, and active information sharing, will be required for them to be effective. Humans will remain a necessary part of securing the software supply chain, also just due to regulations and policies that require their involvement. Ensuring these approaches are user-friendly will help drive broader acceptance across the industry and widespread adoption. In the future, more advanced phishing and spear-phishing attacks using large language models to craft highly convincing messages will be a rising threat, targeting one of the most vulnerable elements to compromise the software supply chain: humans. How can humans keep up? Possible

approaches could include improving training, more restricted environments with fewer privileges, or even deploying LLMs as a counter-defense approach. Looking ahead, as organizations strengthen technological defenses like dependency management and build security, insider threats will likely persist as a major risk. The increasing complexity of internal systems, coupled with limited detection measures and threat models, results in the need for more advanced behavioral monitoring and adaptive security protocols for potential internal threats.

## 6 Other Software Supply Chain Directions

In this section, we discuss cross-cutting topics in software supply-chain security that are not directly related to the three attack vectors.

### 6.1 SBOM

In May 2021, the White House EO 14028 [5] declared the generation of a SBOM as a mandatory artifact for critical software sold to the US government. An SBOM is a nested inventory of "ingredients" that make up the software component or product and helps to identify and keep track of third-party components of a software system [261] to provide transparency and visibility. The primary goal of the SBOM is to prevent attacks such as those targeting SolarWinds [204] and Log4j [7] and to enable faster reaction when an attack occurs. SBOM transparency aids in making informed purchasing decisions. SBOM provides essential risk information, such as licenses, dependencies, vulnerabilities, and misconfigurations, so that organizations can understand the legal and security exposure of the software. Software developers need to understand where particular security vulnerabilities are located in the dependency tree. Most importantly, SBOMs allow companies to control risk by enabling earlier identification and mitigation of vulnerable systems or license-infringing source code.

Despite its many benefits, SBOM generation still faces challenges with automation, particularly in handling the large number of components that are typically part of modern software [72]. State-of-the-art SBOM generation tools are lacking in generating accurate SBOMs [175, 259]. That is, current SBOMs do not contain the exact set of dependencies in the software and frequently report a wrong dependency version [175]. Research is striving to understand which are the current problems in SBOM generation [13, 17, 38, 176, 259] to achieve a complete and correct SBOM.

*6.1.1 Practitioner Challenges.* While SBOMs have received notable policy attention from the US government, software producers have yet to implement them fully. The Linux Foundation's SBOM readiness survey [128] in 2022 on 412 worldwide organizations showed gaps in familiarity with, production planning for, and consumption of SBOMs.

In our Summits, practitioners consistently express a desire to leverage the SBOM for security [53]. By this, they mean they do not want SBOMs only to be generated for compliance reasons but to aid in the security of their products. Industry practitioners stated the benefits of inventory disambiguation and believe that SBOMs can help establish customer trust by providing increased transparency and integrity in their deliverables. Customers can look for extraneous content and identify unwanted third-party dependencies.

However, most of the discussions about SBOM revolved around concerns. There were comments and concerns that the EO only stated that SBOMs should be generated and did not describe what should be done with that data or provide an articulated problem statement for which SBOMs are a solution. One of the biggest hurdles of SBOMs is the overall immaturity of tooling relative to SBOM consumption. Most industrial participants felt the current state of SBOM was of a "compliance-check-the-box" [226]. Stalnaker et al. [208] found that practitioners are concerned about the future of SBOMs becoming more than a minimal compliance requirement. A Summit participant did an

audit of available SBOMs. They noted that nearly none met the **National Telecommunications and Information Administration (NTIA)** minimum requirements [152].

Both industry and government practitioners cited questions about the mechanics of sharing SBOMs. SBOMs can be large. Additionally, if multiple groups within an organization or multiple government agencies use a product, how many times does the SBOM need to be shared? In a 2024 summit, sharing seemed to be evolving to a commitment that the software producer could rapidly share/send an SBOM if asked for it rather than proactively sending the SBOM for all products.

Finally, practitioners are concerned that the use of an SBOM to identify vulnerabilities in the components will yield many vulnerabilities in areas of the components that are not used by the product and, therefore, not reachable or exploitable. In 2021, NTIA introduced **Vulnerability Exploitability Exchange (VEX)** [35]. VEX [35] is a form of security advisory in a standardized, machine-readable format used to convey information about the exploitability of vulnerabilities in software products in the context of the product in which they are used. However, in the Summits, practitioners indicated the lack of adoption of VEX due to concerns about self-declared exploitability statements rather than automated, trustable, objective means for declaring whether a vulnerability was exploitable in the context of the product.

*6.1.2 Generation.* Issues faced during SBOM generation include deficiencies in generated SBOMs, lack of generalizability of tools, incompatibility in standards, and privacy concerns [20, 38, 107, 208]. After more than 2 years and attention from the industry, many tools to produce SBOMs have been created with widely different outputs [168]. Quality expectations for SBOMs have not been agreed upon.

Causes of low-quality SBOMs have been investigated, focusing on single ecosystems. Software ecosystems vary greatly in their approaches to dependency management. Researchers have adopted language-specific methodologies to study SBOM production, uncovering unique issues related to each language [13, 175]. For example, Rabbi et al. [175] presented a quantitative and qualitative comparison of four SBOM generation tools: `ORT`, `cnn`, `syft`, and `cdxgen` and evaluated using 50 popular open source npm projects. The analysis highlights `cnn` as the most reliable tool, providing component names, versions, and dependencies, making it suitable for software supply chain security. In contrast, tools like `ORT` and `syft` lack dependency reporting, while `cdxgen`'s inconsistent performance raises concerns about its reliability in real-world applications. However, the cross-language SBOMs—those created for programs utilizing multiple languages—remain an under-explored area. Moreover, SBOM generation tools implement custom heuristics to solve challenges posed by dependency-solving approaches. A differential analysis of SBOMs generated by four widely used tools across seven software ecosystems highlights significant challenges in identifying dependencies [259].

*6.1.3 Consumption.* Currently, SBOMs are produced but largely not consumed [265]. How might they be consumed to aid security? Consider an application $A$ with an SBOM $SBOM_A$ and a **vulnerability database feed (*VDB*)** that contains information about known vulnerabilities in software and software dependencies. In an ideal world, a software consumer runs a function $f(SBOM_A, VDB)$ to learn if application $A$ is vulnerable to a known vulnerability, likely in one of its dependencies. Ideally, this function would identify whether the dependency with the vulnerability was used, how it was used, and whether this specific use is exploitable. Using this information, the software consumer can put in place mitigation or update to a newer version of the application $A$—or attest that the vulnerability is not security-relevant for the application, such as through a VEX [35] statement. Mitigation, such as selective sandboxing of dependencies, can be particularly effective if the application's source code is unavailable, patches are unavailable, or patch deployment is logistically challenging. The developer of application $A$ can also run a similar process for all of

the application $A$'s dependencies and proactively update dependency versions or apply patches as needed.

Unfortunately, this ideal world is not reality. The current process is messy and error-prone, exacerbated by the low quality of SBOMs and missing data [175]. As a result, even if the software consumer attempted to consider $f(SBOM_A, VDB)$, taking action on the results would cause significant manual effort for both themselves and the developer of application $A$, while not actually improving security in most cases [17].

Missing dynamic components inside of the SBOM and lack of SBOM integration in security pipelines are limiting factors for adoption. Attacks such as Log4Shell actively exploited Java dynamic class loading. Sharma et al. [194] show that a complete SBOM effectively mitigates this kind of attack.

*6.1.4 Sharing.* Another reason for the delay in SBOM adoption is privacy concerns. Companies are worried that exposing SBOMs for their software will also provide attackers with information about potential entry points. Attendees of the software supply chain security summit discussed different approaches to overcome the privacy disclosure problem, e.g., recipient-based SBOMs containing partial information. Xia et al. [249] proposed a blockchain-based solution using verifiable credentials. However, they also acknowledge that blockchain is another limiting factor for adoption and that this technology faces scalability challenges in real-world scenarios. Privacy is not the only obstacle in SBOM sharing. If vendors provide SBOM only in a single format, the software producer has to manage information from different formats to a single SBOM. This operation is not automated and error-prone, posing a barrier to SBOM adoption [265].

*6.1.5 VEX Use.* VEX [35] information can be represented inside an existing SBOM, or in a dedicated VEX addendum. This addendum is a security advisory indicating the status of known vulnerabilities affecting a product (or products). The VEX could greatly improve the SBOM efficacy for security. A developer can include a VEX statement to communicate to a customer that their application is not vulnerable to the vulnerability in the vulnerable dependency. However, from the software supply chain Summits emerged a consensus that the manual nature of creating VEX information will significantly limit its value [261].

Integrating static and dynamic analysis tools [62, 169] into the build process may provide the necessary automation for the production of VEX. By feeding tools with reliable and detailed information about vulnerabilities and their specifics, such as where the vulnerability is located in the dependency and what kind of data may trigger it, tools can test whether the vulnerability is reachable/exploitable. However, manual effort is currently required to identify which specific portion of the code contains the vulnerability and how to reach that part. Security patches contain useful information that can be used to identify vulnerable elements in the code and how they are triggered. However, this information was largely ignored. In 2024, Dunlap et al. [51] used security patches with LLM models to map security advisories to their vulnerable functions. Moreover, the automated production of proper VEX may enable more advanced vulnerability mitigation techniques, such as additional input sanitation or local sandboxing [235].

Novel techniques are needed to make the production of VEX automated, accurate, and trustable. Future research can focus on automating dependency selection by including information on vulnerability and other security metrics. Doing so requires higher-quality vulnerability, metadata, and code information. Key research directions include prioritizing the selection of dependencies, using techniques to extract dependency information automatically, and building practical tools for presenting this data to developers.

### 6.2 Practitioner Challenge: Self-Attestation and Provenance

The US EO requires the **chief executive officer (CEO)** of government contractors to submit a form attesting to (1) conformity with secure software development practices; and (2) the integrity and provenance of open source software used within any portion of a product. The form of the self-attestation is administered through and the US National Coordinator for Critical Infrastructure Security and Release and the Office of Management and Budget.[2] In the software supply chain context, provenance refers to not only the identity that created each dependency and transitive dependency but also the process through which each software component was built. For example, provenance is a key part of the SLSA framework [161], and systems such as in-toto[3] [225] can be used to capture and communicate provenance information. Industrial participants shared concerns about ambiguity in self-attestation requirements. However, many participants agreed that the SSDF and attestation are far more foundational to security than the SBOM. However, according to Summit attendees, concerns about the burden of a CEO attesting the specifications have caused organizations to deploy their products as open source, since open source software is exempt from self-attestation.

### 6.3 LLM and Software Supply Chains

LLMs are increasingly leveraged across the software development lifecycle for different tasks. Hence, LLMs have become a part of the software supply chain, similar to code dependencies, build infrastructure, and humans. Several prior works have leveraged LLMs for different software tasks, including code generation [122], vulnerability detection [127, 133, 172, 270], malicious package detection [263] and even exploring LLMs as potential attack vectors [149, 195, 252]. Huang et al. [95] performed a literature review and provided an overview of the roles of stakeholders, incorporated components, and various types of risk associated with the LLM supply chain. The study identified four primary categories of LLM supply chain risks: security, privacy, delivery, and legal risks. Furthermore, the authors developed a taxonomy of these risks and proposed corresponding mitigation strategies. In another study, Balayn et al. [12] interviewed 71 participants from 10 private organizations to analyze stakeholders' perceptions of AI explainability and transparency. The study highlighted stakeholders' information needs and the challenges they encounter, emphasizing the importance of considering the "who," "what," and "why" in **explainable AI (XAI)** practices. Additionally, the research offers methodological guidance and lessons for future studies focusing on (XAI). In this section, we discussed the challenges practitioners face when using LLMs, current research leveraging LLMs, and future research directions for LLMs.

*6.3.1 Practitioner Challenges.* Within the last year, LLM-based systems, such as ChatGPT, have been increasingly used for automated code generation, testing, program repair, and summarization. According to a recent industry summit [229], participants stated that generative AI is in use to reduce developers' workload by fixing easy vulnerabilities. Participants also mentioned that LLMs could resolve 7.5% of reviewer comments automatically. However, participants also expressed concerns that the public's accelerated use of LLM can lead to large-scale data exfiltration. Practitioners frequently pull LLM output into their product and contribute their own proprietary data into LLM's training phase through their queries. As a result, despite generative AI's benefits, companies hesitate to adopt LLMs due to trust issues, particularly concerns about hallucinations and insecure coding impact on development.

[2] https://www.cisa.gov/resources-tools/resources/secure-software-development-attestation-form
[3] https://in-toto.io/

*6.3.2 Code Generation.* LLMs such as GitHub Copilot[4] and CodeGeeX[5] have revolutionized the coding tasks including code completion [80, 134], code evaluation [89, 279], repair [103, 157], implementing encryption schemes [61], and code translation [116, 216]. While LLMs are capable of code generation, they also suffer from insecure code generation [31, 34, 243]. Existing models can unintentionally introduce security flaws or generate code that does not comply with the required standards [102, 255] or even actually contain security vulnerabilities [124, 167]. GitHub Copilot-generated 40% of programs had security vulnerabilities [167], with similar issues noted in other state-of-the-art LLMs [65, 124, 148]. However, compared to commonly utilized platforms by developers, such as Stack Overflow, current results indicate that LLMs might produce less vulnerable code [82].

LLMs often rely on third-party dependencies to enhance functionality. However, the LLM's supply chain may contain compromised components that introduce vulnerabilities into the final models, leading to potential LLM supply chain security risk [114]. Additionally, since LLMs are trained on public and open repositories, these models can unintentionally propagate existing vulnerabilities by replicating insecure patterns and selecting vulnerable dependencies [10]. However, both industry and academia have contributed to mitigating the security risks. Proposed solutions include frameworks [85, 141] and AI tools [2, 3] specifically designed to reduce these risks and address security concerns effectively.

*6.3.3 Vulnerability Detection.* Prior research has shown that LLMs can be more effective than standard static analysis techniques for predicting software vulnerabilities with higher accuracy [11, 23, 105, 263, 270]. Singla et al. [201] analyzed previous security failure reports using LLM and achieved an average accuracy of 68% in categorizing breaches. Khare et al. [106] compared the performance of LLMs (GPT-4 and CodeLlama) to deep learning models (DeepDFA and LineVul), finding that LLMs demonstrated similar accuracy. However, deep-learning models have limitations in real-world applications, where LLMs offer better interpretability and explanation. Additionally, open source LLMs have been utilized to better align security advisories with vulnerable functions. The approach shows a 173% increase in precision with only an 18% decrease in recall compared to automated methods [51]. This improvement in precision supports more accurate security analysis and threat mitigation. Then studies have compared the effectiveness of LLMs in vulnerability detection with conventional methods and discovered that transformer-based LLMs surpass not only traditional static tools [93, 132, 250, 276] but also exceed the performance of deep learning models [68, 221]. Frameworks with the integration of deep learning and LLM are capable of achieving 92.65% accuracy [158]. However, for certain languages such as Java, LLMs do not exceed the performance of the existing tools [33]. Moreover, issues like false positives [275] and hallucinations [207], unconstrained training datasets, and complex model parameters diminish the performance of LLMs. To overcome the challenges, recent studies proposed different methodologies and frameworks that leverage LLMs to enhance vulnerability detection across different programming contexts, outperforming traditional or static analysis tools [47, 73, 125].

*6.3.4 LLMs as Attack Vector.* In software supply chain security, LLMs are considered a prominent source of introducing new vulnerabilities that can be exploited as attack vectors, which requires a thorough review during the LLMs training phase [30, 257]. LLMs can be utilized as an attack vector through prompt injection [131], Malicious Programming Prompt [87] and third-party APIs [278] integration. Prior research studies highlight the methods and implications of these attacks [230, 253]. For example, the Sandwich attack [230], showed how LLMs can be exploited as attack vectors,

[4]https://github.com/features/copilot
[5]https://codegeex.cn/en-US

allowing adversaries to manipulate state-of-the-art models into generating harmful or misaligned responses. Although LLMs can be used for vulnerability management and automate the analysis of security failure, the models also pose risks due to their reliance on potentially insecure code, data quality issues, and privacy concerns [241]. Research demonstrated that LLMs can effectively generate tactics, techniques, and procedures for interpreted malware [277] and security tests [276], highlighting LLMs potential to both identify and exploit vulnerabilities in systems' dependencies.

*6.3.5 LLM's Performance and Scalability.* While ongoing research is necessary to effectively leverage LLMs in areas like code generation, vulnerability detection, and mitigating LLM attack vectors, future studies should also focus on improving LLM performance holistically, enabling their application across a wide range of domains beyond just software supply chain security. For example, prior research often highlights the high cost associated with using LLMs [263], especially for large-scale analysis. LLMs also suffer in analyzing large files and modules due to inherent limitations in token usage and contextual understanding and their ability to maintain context across separate segments of a large file and modules. Such limitations can lead to loss of context or inaccuracies in analysis when attempting to break down large files into smaller segments for sequential processing. Interdisciplinary research is needed to improve the efficiency and cost-effectiveness of LLMs in handling large files while ensuring interoperability across different modules. Future research also needs to focus on mode collapse [197] and hallucination [100] to improve LLM performance [25, 197].

### 6.4 Practitioner Challenge: Supply Chain Standards, Guidelines, and Frameworks

Software security standards, guidelines, and frameworks, including 800-218 SSDF [150], NIST 800-161 [151], SLSA [161], and S2C2F [162], have emerged to guide what organizations can do to reduce software supply chain risk. Compliance with the EO requires attestation to most SSDF tasks, making the SSDF an important standard. The other most mentioned framework used by participants was SLSA [161] security framework. Some practitioners still have difficulty deciding what to be guided by with the numerous standards, guidelines, and frameworks available. The P-SSCRM [246] provides the union of nine software supply chain frameworks and a mapping between these nine frameworks.

### 6.5 Software Supply Chain Security Measurement

The US Federal Cybersecurity Research and Development Strategic Plan [90] highlights software security measurements as a research priority. However, effectively measuring software security remains a complex challenge due to the unpredictable and context-dependent nature of software systems [32, 91, 189]. At present, development teams are making security-critical dependency selection decisions under time pressure and with limited information. Identifying, aggregating and disseminating useful software supply chain security metrics and insights from the sources described above could support practitioners in their assessment and selection needs. We have identified three requirements for providing this support; metrics curation, data publication, and periodic review. Metrics curation requires evaluating industry experience and research progress to identify the data and metrics most relevant to practitioner decision-making in software supply chain security. Data publication involves collecting and presenting the curated data so that software development teams can apply research findings in their own contexts. Providing current data and metrics for individual dependencies and their ecosystems moves analysis from being research-focused to being practice-focused. Periodic review is driven by the dynamic nature of the supply chain and the attacks against it. As the problems of and solutions for protecting the supply chain evolve, "state of

the practice" advice must also evolve. We plan a "Software Supply Chain Security Dashboard" to accommodate practitioner data needs and an annual report to address the need for periodic review.

## 7 Conclusions

Software supply chain security is a diverse and challenging topic that requires sustained attention and effort from both industry and academia. The last several years have seen an explosion of interest following the SolarWinds [60, 204] and log4j [7] incidents. In this article, we have described three key attack vectors: (1) code dependencies (including both intentional and unintentional vulnerabilities), (2) build infrastructure, and (3) humans. For each attack vector, we detailed significant efforts by both industry and academia. For example, in industry, SCA tools, SBOMs, and OpenSSF Scorecard are gaining traction to address code dependencies and SLSA, in-toto, and TUF are helping to secure build processes. Academic research has contributed to these efforts (e.g., originally developing in-toto and TUF). More primarily and the focus of our own research, academic research has developed novel analysis tools (e.g., to enhance vulnerability information and discover vulnerabilities in build configuration), performed empirical studies that have measured existing tools (e.g., SCA tools) and ecosystems (e.g., npm, PyPI, GitHub Actions), and interviewed practitioners to gain deeper insight into underlying challenges and motivations to adopt solutions.

While significant progress has been made, our summits with industry and government highlight the need for continued research. Managing vulnerabilities in dependencies is *hard*: vulnerability information is messy, and practitioners spend significant time both determining if code dependencies need to be updated and updating them. Developers also need help *choosing* dependencies such that they avoid both malicious libraries (e.g., on npm and PyPI) as well as under-maintained and abandoned libraries. The build side of the supply chain has equally hard challenges. Legacy build environments are pervasive and hard to change. We simultaneously lack robust analysis tools for the code and configuration that builds software, and the ability to retrofit build environments to provide provenance and transparency. Software projects are also riddled with non-determinism in both project code and build code, preventing build reproducibility.

The recent xz-utils attack [64, 84] is also a stark reminder that current efforts have only touched the surface of the problem. Most notably, it is unclear if *any* existing security practice could have meaningfully prevented or even *detected* the xz-utils attack. Paying careful attention to the human factor in the software supply chain is as important now as ever. Simultaneously, the use of LLMs for both code generation and application functionality is introducing new risks, highlighting the need to treat LLM models as component in the software supply chain. However, LLMs have their own nuances that will require novel threat models and novel solutions.

## References

[1] GitHub. 2022. Embedded Malicious Code in node-ipc. Retrieved March 16, 2022 from https://github.com/advisories/GHSA-97m3-w2cp-4xx6

[2] Codeium. 2018. Retrieved from https://codeium.com/blog/code-security-chatgpt-issues

[3] TabNine. 2018. AI Code Completions. Retrieved from https://github.com/codota/TabNine

[4] Socket, Inc. 2022. Retrieved December 2, 2023 from https://socket.dev/

[5] Federal Register. Executive Order 14028: Improving the Nation's Cybersecurity. Retrieved May 12, 2021 from https://www.federalregister.gov/documents/2021/05/17/2021-10460/improving-the-nations-cybersecurity

[6] William Enck Yasemin Acar, Michel Cucker, Alexandros Kapravelos, Christian Kastner, and Laurie Williams. June 2023. S3C2 summit 2023-06: Government secure supply chain summit. arXiv: 2308.06850. Retrieved from https://arxiv.org/abs/2308.06850

[7] Cybersecurity & Infrastructure Security Agency. 2022. Apache Log4j Vulnerability Guidance. Retrieved April 08, 2022 from https://www.cisa.gov/news-events/news/apache-log4j-vulnerability-guidance

[8] Mahmoud Alfadel, Diego Elias Costa, Emad Shihab, and Bram Adams. 2023. On the discoverability of npm vulnerabilities in node. js projects. *ACM Transactions on Software Engineering and Methodology* 32, 4 (2023), 1–27.

[9] Cloud Security Alliance. 2024. Global Security Database (GSD). Retrieved from https://github.com/cloudsecurityalliance/gsd-database
[10] Owura Asare, Meiyappan Nagappan, and N. Asokan. 2023. Is GitHub's copilot as bad as humans at introducing vulnerabilities in code? *Empirical Software Engineering* 28, 6 (2023), 129.
[11] Virendra Ashiwal, Soeren Finster, and Abdallah Dawoud. 2024. LLM-based vulnerability sourcing from unstructured data. In *2024 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, 634–641.
[12] Agathe Balayn, Lorenzo Corti, Fanny Rancourt, Fabio Casati, and Ujwal Gadiraju. 2024. Understanding stakeholders' perceptions and needs across the LLM supply chain. arXiv preprint arXiv:2405.16311. Retrieved from https://arxiv.org/abs/2405.16311
[13] Musard Balliu, Benoit Baudry, Sofia Bobadilla, Mathias Ekstedt, Martin Monperrus, Javier Ron, Aman Sharma, Gabriel Skoglund, César Soto-Valero, and Martin Wittlinger. 2023. Challenges of producing software bill of materials for Java. *IEEE Security & Privacy* 21, 6 (Nov. 2023), 12–23.
[14] Frederick Barr-Smith, Tim Blazytko, Richard Baker, and Ivan Martinovic. 2022. Exorcist: Automated differential analysis to detect compromises in closed-source software supply chains. In *Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*, 51–61.
[15] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. 2013. The evolution of project inter-dependencies in a software ecosystem: The case of Apache. In *2013 IEEE International Conference on Software Maintenance*. IEEE, 280–289.
[16] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. 2015. How the Apache community upgrades dependencies: An evolutionary study. *Empirical Software Engineering* 20 (2015), 1275–1317.
[17] Giacomo Benedetti, Serena Cofano, Alessandro Brighente, and Mauro Conti. 2024. The impact of SBOM generators on vulnerability assessment in Python: A comparison and a novel approach. arXiv:2409.06390. Retrieved from https://arxiv.org/abs/2409.06390
[18] Giacomo Benedetti, Luca Verderame, and Alessio Merlo. 2022. Automatic security assessment of GitHub actions workflows. In *Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses (SCORED '22)*. ACM, New York, NY, 37–45. DOI: https://doi.org/10.1145/3560835.3564554
[19] Sebastian Benthall. 2017. Assessing software supply chain risk using public data. In *2017 IEEE 28th Annual Software Technology Conference (STC)*. IEEE, 1–5.
[20] Tingting Bi, Boming Xia, Zhenchang Xing, Qinghua Lu, and Liming Zhu. 2024. On the way to SBOMs: Investigating design issues and solutions in practice. *ACM Transactions on Software Engineering and Methodology* 33, 6 (2024), 1–25.
[21] Jennifer Blackhurst, M. Johnny Rungtusanatham, Kevin Scheibe, and Saurabh Ambulkar. 2018. Supply chain vulnerability assessment: A network based visualization and clustering analysis approach. *Journal of Purchasing and Supply Management* 24, 1 (2018), 21–30.
[22] Christopher Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. 2016. How to break an API: Cost negotiation and community values in three software ecosystems. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 109–120.
[23] Biagio Boi, Christian Esposito, and Sokjoon Lee. 2024. Smart contract vulnerability detection: The role of large language model (LLM). *ACM SIGAPP Applied Computing Review* 24, 2 (2024), 19–29.
[24] Lina Boughton, Courtney Miller, Yasemin Acar, Dominik Wermke, and Christian Kästner. 2024. Decomposing and measuring trust in open-source software supply chains. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering: New Ideas and Emerging Results (IEEE/ACM ICSE-NIER '24)*. IEEE/ACM.
[25] Martin Briesch, Dominik Sobania, and Franz Rothlauf. 2023. Large language models suffer from their own output: An analysis of the self-consuming training loop. arXiv:2311.16822. Retrieved from https://arxiv.org/abs/2311.16822
[26] Aline Brito, Laerte Xavier, Andre Hora, and Marco Tulio Valente. 2018. Why and how Java developers break APIs. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 255–265.
[27] Simon Butler, Jonas Gamalielsson, Björn Lundell, Christoffer Brax, Anders Mattsson, Tomas Gustavsson, Jonas Feist, Bengt Kvarnström, and Erik Lönroth. 2023. On business adoption and use of reproducible builds for open and closed source software. *Software Quality Journal* 31, 3 (2023), 687–719.
[28] Yulu Cao, Lin Chen, Wanwangying Ma, Yanhui Li, Yuming Zhou, and Linzhang Wang. 2022. Towards better dependency management: A first look at dependency smells in Python projects. *IEEE Transactions on Software Engineering* 49, 4 (2022), 1741–1765.
[29] Ramaswamy Chandramouli, Frederick Kautz, and Santiago Torres-Arias. 2024. Strategies for the integration of software supply chain security in DevSecOps CI/CD pipelines. NIST Special Publication 800-204D. Retrieved February 2024 from https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-204D.pdf

[30] Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, et al. 2024. A survey on evaluation of large language models. *ACM Transactions on Intelligent Systems and Technology* 15, 3 (2024), 1–45.

[31] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. arXiv:2107.03374. Retrieved from https://arxiv.org/abs/2107.03374

[32] Yi Cheng, Julia Deng, Jason Li, Scott A. DeLoach, Anoop Singhal, and Xinming Ou. 2014. Metrics of security. In *Cyber Defense and Situational Awareness*. Springer, 263–295.

[33] Anton Cheshkov, Pavel Zadorozhny, and Rodion Levichev. 2023. Evaluation of ChatGPT model for vulnerability detection. arXiv:2304.07232. Retrieved from https://arxiv.org/abs/2304.07232

[34] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2023. PaLM: Scaling language modeling with pathways. *Journal of Machine Learning Research* 24, 240 (2023), 1–113.

[35] CISA. 2022. Vulnerability Exploitability eXchange (VEX). Retrieved from https://www.cisa.gov/sites/default/files/publications/VEX_Use_Cases_Document_508c.pdf

[36] Cloud Native Computing Foundation. 2022. Software Supply Chain Best Practices (SSCP). Retrieved from https://project.linuxfoundation.org/hubfs/CNCF_SSCP_v1.pdf

[37] CNCF. Cloud Native Computing Foundation (CNCF). Retrieved September 26, 2024 from https://www.cncf.io/

[38] Serena Cofano, Giacomo Benedetti, and Matteo Dell'Amico. 2024. SBOM generation tools in the Python ecosystem: An in-detail analysis. arXiv:2409.01214. Retrieved from https://arxiv.org/abs/2409.01214

[39] Joel Cox, Eric Bouwers, Marko van Eekelen, and Joost Visser. 2015. Measuring dependency freshness in software systems. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2, 109–118. DOI: https://doi.org/10.1109/ICSE.2015.140

[40] Roland Croft, M. Ali Babar, and M. Mehdi Kholoosi. 2023. Data quality for software vulnerability datasets. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 121–133.

[41] DataDog. 2022. GuardDog. Retrieved from https://github.com/datadog/guarddog

[42] Alexandre Decan, Tom Mens, and Eleni Constantinou. 2018. On the evolution of technical lag in the npm package dependency network. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 404–414. DOI: https://doi.org/10.1109/ICSME.2018.00050

[43] Alexandre Decan, Tom Mens, and Eleni Constantinou. 2018. On the impact of security vulnerabilities in the npm package dependency network. In *15th International Conference on Mining Software Repositories*, 181–191.

[44] Alexandre Decan, Tom Mens, and Hassan Onsori Delicheh. 2023. On the outdatedness of workflows in the GitHub Actions ecosystem. *Journal of Systems and Software* 206 (2023), 111827.

[45] Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. 2017. Keep me updated: An empirical study of third-party library updatability on Android. In *2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, New York, NY, 2187–2200. DOI: https://doi.org/10.1145/3133956.3134059

[46] Jens Dietrich, David Pearce, Jacob Stringer, Amjed Tahir, and Kelly Blincoe. 2019. Dependency versioning in the wild. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 349–359.

[47] Xueying Du, Geng Zheng, Kaixin Wang, Jiayi Feng, Wentai Deng, Mingwei Liu, Bihuan Chen, Xin Peng, Tao Ma, and Yiling Lou. 2024. Vul-RAG: Enhancing LLM-based vulnerability detection via knowledge-level rag. arXiv:2406.11147. Retrieved from https://arxiv.org/abs/2406.11147

[48] Ruian Duan, Omar Alrawi, Ranjita Pai Kasturi, Ryan Elder, Brendan Saltaformaggio, and Wenke Lee. 2020. Towards measuring supply chain attacks on package managers for interpreted languages. arXiv:2002.01139. Retrieved from https://arxiv.org/abs/2002.01139

[49] Trevor Dunlap, Yasemin Acar, Michel Cucker, William Enck, Alexandros Kapravelos, Christian Kastner, and Laurie Williams. 2023. S3C2 summit 2023-02: Industry secure supply chain summit. arXiv:2307.16557. Retrieved from http://arxiv.org/abs/2307.16557

[50] Trevor Dunlap, Elizabeth Lin, William Enck, and Bradley Reaves. 2023. VFCFinder: Seamlessly pairing security advisories and patches. arXiv:2311.01532. Retrieved from https://arxiv.org/abs/2311.01532

[51] Trevor Dunlap, John Speed Meyers, Bradley Reaves, and William Enck. 2024. Pairing security advisories with vulnerable functions using open-source LLMs. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 350–369.

[52] Trevor Dunlap, Seaver Thorn, William Enck, and Bradley Reaves. 2023. Finding fixed vulnerabilities with off-the-shelf static analysis. In *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*. IEEE, 489–505.

[53] William Enck and Laurie Williams. 2022. Top five challenges in software supply chain security: Observations from 30 industry and government organizations. *IEEE Security & Privacy* 20, 2 (2022), 96–100. DOI: https://doi.org/10.1109/MSEC.2022.3142338

[54] Datadog Engineering. 2023. Secure publication of datadog agent integrations with TUF and in-toto. *Datadog Engineering Blog*. Retrieved September 26, 2024 from https://www.datadoghq.com/blog/engineering/secure-publication-of-datadog-agent-integrations-with-tuf-and-in-toto/
[55] Equifax. 2017. Equifax releases details on cybersecurity incident, announces personnel changes. Retrieved September 24, 2024 from https://investor.equifax.com/news-and-events/news/2017/09-15-2017-224018832
[56] EU 2022. EU Cyber Resilience Act. Retrieved from https://digital-strategy.ec.europa.eu/en/library/cyber-resilience-act
[57] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. AC/C++ code vulnerability dataset with code changes and CVE summaries. In *17th International Conference on Mining Software Repositories*, 508–512.
[58] Clarisse Feio, Nuno Santos, Nelson Escravana, and Bernardo Pacheco. 2024. An empirical study of DevSecOps focused on continuous security testing. In *2024 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, 610–617.
[59] Gabriel Ferreira, Limin Jia, Joshua Sunshine, and Christian Kästner. 2021. Containing malicious package updates in npm with a lightweight permission system. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1334–1346.
[60] FireEye. December 13, 2020. Highly evasive attacker leverages solarwinds supply chain to compromise multiple global victims with SUNBURST backdoor. Retrieved from https://cloud.google.com/blog/topics/threat-intelligence/evasive-attacker-leverages-solarwinds-supply-chain-compromises-with-sunburst-backdoor/
[61] Ehsan Firouzi and Mohammad Ghafari. 2024. Time to separate from StackOverflow and match with ChatGPT for encryption. *Journal of Systems and Software* (2024), 112135.
[62] Darius Foo, Jason Yeo, Hao Xiao, and Asankhaya Sharma. 2019. The dynamics of software composition analysis. arXiv:1909.00973 (2019). Retrieved from https://arxiv.org/abs/1909.00973
[63] Marcel Fourné, Dominik Wermke, William Enck, Sascha Fahl, and Yasemin Acar. 2023. It's like flossing your teeth: On the importance and challenges of reproducible builds for software supply chain security. In *44th IEEE Symposium on Security and Privacy*.
[64] Andres Freund. 2024. Backdoor in upstream xz/liblzma leading to ssh server compromise. Retrieved March 29, 2024 from https://www.openwall.com/lists/oss-security/2024/03/29/4
[65] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. arXiv:2204.05999. Retrieved from https://arxiv.org/abs/2204.05999
[66] Fabian Froh, Matías Gobbi, and Johannes Kinder. 2023. Differential static analysis for detecting malicious updates to open source packages. In *Proceedings of the 2023 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*, 41–49.
[67] Gal Ofri. 2023. SLSA Provenance Blog Series, Part 3: The Challenges of Adopting SLSA Provenance. Retrieved from https://www.legitsecurity.com/blog/slsa-provenance-blog-series-part3-challenges-of-adopting-slsa-provenance
[68] Zeyu Gao, Hao Wang, Yuchen Zhou, Wenyu Zhu, and Chao Zhang. 2023. How far have we gone in vulnerability detection using large language models. arXiv:2311.12420. Retrieved from https://arxiv.org/abs/2311.12420
[69] Kalil Garrett, Gabriel Ferreira, Limin Jia, Joshua Sunshine, and Christian Kästner. 2019. Detecting suspicious package updates. In *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. IEEE, 13–16.
[70] GitHub. 2024. Dependabot: Security Updates for Your Dependencies. Retrieved September 25, 2024 from https://github.com/dependabot
[71] GitHub. 2024. GitHub Advisory Database. Retrieved from https://github.com/advisories
[72] Antonios Gkortzis, Daniel Feitosa, and Diomidis Spinellis. 2021. Software reuse cuts both ways: An empirical analysis of its relationship with security vulnerabilities. *Journal of Systems and Software* 172, 110653 (Feb. 2021), 110653.
[73] José Gonçalves, Tiago Dias, Eva Maia, and Isabel Praça. 2024. SCoPE: Evaluating LLMs for software vulnerability detection. arXiv:2407.14372. Retrieved from https://arxiv.org/abs/2407.14372
[74] Danielle Gonzalez, Thomas Zimmermann, Patrice Godefroid, and Max Schäfer. 2021. Anomalicious: Automated detection of anomalous and potentially malicious commits on GitHub. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 258–267.
[75] Google. 2016. OSS-Fuzz: Continuous Fuzzing for Open Source Software. Retrieved from https://github.com/google/oss-fuzz
[76] Google. 2024. Open Source Vulnerability Database. Retrieved from https://osv.dev/
[77] David Grove and Craig Chambers. 2001. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 23, 6 (2001), 685–746.
[78] Yacong Gu, Lingyun Ying, Huajun Chai, Yingyuan Pu, Haixin Duan, and Xing Gao. 2024. More haste, less speed: Cache related security threats in continuous integration services. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 138–138.

[79] Yacong Gu, Lingyun Ying, Huajun Chai, Chu Qiao, Haixin Duan, and Xing Gao. 2023. Continuous intrusion: Characterizing the security of continuous integration services. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1561–1577.

[80] Daya Guo, Canwen Xu, Nan Duan, Jian Yin, and Julian McAuley. 2023. Longcoder: A long-range pre-trained language model for code completion. In *International Conference on Machine Learning*. PMLR, 12098–12107.

[81] Wenbo Guo, Zhengzi Xu, Chengwei Liu, Cheng Huang, Yong Fang, and Yang Liu. 2023. An Empirical study of malicious code in PyPI ecosystem. arXiv:2309.11021. Retrieved from https://arxiv.org/abs/2309.11021

[82] Sivana Hamer, Marcelo d'Amorim, and Laurie Williams. 2024. Just another copy and paste? Comparing the security vulnerabilities of ChatGPT generated code and StackOverflow answers. In *2024 IEEE Security and Privacy Workshops (SPW)*. IEEE, 87–94.

[83] Sivana Hamer, Nasif Imtiaz, Mahzabin Tamanna, Preya Shabrina, and Laurie Williams. 2024. Trusting code in the wild: Exploring contributor reputation measures to review dependencies in the Rust ecosystem. arXiv:2406.10317. Retrieved from https://arxiv.org/abs/2406.10317

[84] Red Hat. 2024. Urgent security alert for Fedora Linux 40 and Fedora Rawhide users. Retrieved March 29, 2024 from https://www.redhat.com/en/blog/urgent-security-alert-fedora-40-and-rawhide-users

[85] Jingxuan He and Martin Vechev. 2023. Large language models for code: Security hardening and adversarial testing. In *2023 ACM SIGSAC Conference on Computer and Communications Security*, 1865–1879.

[86] Heartbleed. 2021. Heartbleed Bug. Retrieved July 17, 2021 from https://heartbleed.com/

[87] John Heibel and Daniel Lowd. 2024. MaPPing your model: Assessing the impact of adversarial attacks on LLM-based programming assistants. arXiv:2407.11072. Retrieved from https://arxiv.org/abs/2407.11072

[88] Daan Hommersom, Antonino Sabetta, Bonaventura Coppola, Dario Di Nucci, and Damian A Tamburri. 2024. Automated mapping of vulnerability advisories onto their fix commits in open source repositories. *ACM Transactions on Software Engineering and Methodology* 33, 5 (2024), 1–28.

[89] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large language models for software engineering: A systematic literature review. arXiv:2308.10620. Retrieved from https://arxiv.org/abs/2308.10620

[90] The White House. 2023. Federal Cybersecurity Research and Development Strategic Plan 2023. Retrieved from https://www.whitehouse.gov/wp-content/uploads/2024/01/Federal-Cybersecurity-RD-Strategic-Plan-2023.pdf

[91] The White House. 2024. Back to the Building Blocks: A Path Toward Secure and Measurable Software. Retrieved from https://www.whitehouse.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf

[92] US White House. 2021. Executive Order 14028 on Improving the Nation's Cybersecurity. Retrieved May 12, 2021 from https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity/

[93] Jie Hu, Qian Zhang, and Heng Yin. 2023. Augmenting greybox fuzzing with generative AI. arXiv:2306.06782. Retrieved from https://arxiv.org/abs/2306.06782

[94] Cheng Huang, Nannan Wang, Ziyan Wang, Siqi Sun, Lingzi Li, Junren Chen, Qianchong Zhao, Jiaxuan Han, Zhen Yang, and Lei Shi. 2024. DONAPI: Malicious NPM packages detector using behavior sequence knowledge mapping. arXiv:2403.08334. Retrieved from https://arxiv.org/abs/2403.08334

[95] Kaifeng Huang, Bihuan Chen, You Lu, Susheng Wu, Dingji Wang, Yiheng Huang, Haowen Jiang, Zhuotong Zhou, Junming Cao, and Xin Peng. 2024. Lifting the veil on the large language model supply chain: Composition, risks, and mitigations. arXiv:2410.21218. Retrieved from https://arxiv.org/abs/2410.21218

[96] Akinori Ihara, Daiki Fujibayashi, Hirohiko Suwa, Raula Gaikovina Kula, and Kenichi Matsumoto. 2017. Understanding when to adopt a library: A case study on ASF projects. In *Proceedings of the 13th IFIP WG 2.13 International Conference on Open Source Systems: Towards Robust Practices (OSS '17)*. Springer International Publishing, 128–138.

[97] Nasif Imtiaz, Seaver Thorn, and Laurie Williams. 2021. A comparative study of vulnerability reporting by software composition analysis tools. In *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 1–11.

[98] Vipawan Jarukitpipat, Klinton Chhun, Wachirayana Wanprasert, Chaiyong Ragkhitwetsagul, Morakot Choetkiertikul, Thanwadee Sunetnanta, Raula Gaikovina Kula, Bodin Chinthanet, Takashi Ishio, and Kenichi Matsumoto. 2022. V-Achilles: An interactive visualization of transitive security vulnerabilities. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 1–4.

[99] Abbas Javan Jafari, Diego Elias Costa, Emad Shihab, and Rabe Abdalkareem. 2023. Dependency update strategies and package characteristics. *ACM Transactions on Software Engineering and Methodology* 32, 6 (2023), 1–29.

[100] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. 2023. Survey of hallucination in natural language generation. *ACM Computing Surveys* 55, 12 (2023), 1–38.

[101] Peiyang Jia, Chengwei Liu, Hongyu Sun, Chengyi Sun, Mianxue Gu, Yang Liu, and Yuqing Zhang. 2022. Cargo ecosystem dependency-vulnerability knowledge graph construction and vulnerability propagation study. arXiv:2210.07482. Retrieved from https://arxiv.org/abs/2210.07482

[102] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A survey on large language models for code generation. arXiv:2406.00515. Retrieved from https://arxiv.org/abs/2406.00515

[103] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-aware neural machine translation for automatic program repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1161–1173.

[104] Hong Jin Kang, Truong Giang Nguyen, Bach Le, Corina S Păsăreanu, and David Lo. 2022. Test mimicry to assess the exploitability of library vulnerabilities. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 276–288.

[105] Evangelos Katsadouros, Charalampos Z. Patrikakis, and George Hurlburt. 2023. Can large language models better predict software vulnerability? *IT Professional* 25, 3 (2023), 4–8.

[106] Avishree Khare, Saikat Dutta, Ziyang Li, Alaia Solko-Breslin, Rajeev Alur, and Mayur Naik. 2023. Understanding the effectiveness of large language models in detecting security vulnerabilities. arXiv:2311.16169. Retrieved from https://arxiv.org/abs/2311.16169

[107] Berend Kloeg, Aaron Yi Ding, Sjoerd Pellegrom, and Yury Zhauniarovich. 2024. Charting the path to SBOM adoption: A business stakeholder-centric approach. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, 1770–1783.

[108] Tadayoshi Kohno, Yasemin Acar, and Wulf Loh. 2023. Ethical frameworks and computer security trolley problems: Foundations for conversations. In *32nd USENIX Security Symposium (USENIX Security '23)*, 5145–5162.

[109] Igibek Koishybayev, Aleksandr Nahapetyan, Raima Zachariah, Siddharth Muralee, Bradley Reaves, AlexandrosKapravelos, and AravindMachiry. 2022. Characterizing the security of GitHub CI workflows. In *31st USENIX Security Symposium (USENIX Security '22)*. USENIX Association, 2747–2763. Retrieved from https://www.usenix.org/conference/usenixsecurity22/presentation/koishybayev

[110] Alexander Krause, Jan H. Klemmer, Nicolas Huaman, Dominik Wermke, Yasemin Acar, and Sascha Fahl. 2023. Pushed by accident: A mixed-methods study on strategies of handling secret information in source code repositories. In *32nd USENIX Security Symposium (USENIX '23)*. USENIX Association. Retrieved August 9, 2023 from https://www.usenix.org/conference/usenixsecurity23/presentation/krause

[111] Kubernetes and IBM. 2024. Building an Image Trust Service on Kubernetes with Notary and TUF. Retrieved September 26, 2024 from https://kubernetes.io/case-studies/ibm/

[112] Philipp Kuehn, Markus Bayer, Marc Wendelborn, and Christian Reuter. 2021. OVANA: An approach to analyze and improve the information quality of vulnerability databases. In *16th International Conference on Availability, Reliability and Security*, 1–11.

[113] Raula Gaikovina Kula, Daniel M German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. 2018. Do developers update their library dependencies? An empirical study on the impact of security advisories on library migration. *Empirical Software Engineering* 23 (2018), 384–417.

[114] Varun Kumar. Aug 11, 2024. Software Supply Chain Vulnerabilities in Large Language Models (LLMs). Retrieved from https://www.practical-devsecops.com/software-supply-chain-vulnerabilities-llms

[115] Datadog Security Labs. 2023. Open-Source Dataset of Malicious Software Packages. Retrieved December 2, 2023 from https://github.com/datadog/malicious-software-packages-dataset

[116] Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. arXiv:2006.03511. Retrieved from https://arxiv.org/abs/2006.03511

[117] P. Ladisa, H. Plate, M. Martinez, and O. Barais. 2023. SoK: Taxonomy of attacks on open-source software supply chains. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, Los Alamitos, CA, 1509–1526. DOI: https://doi.org/10.1109/SP46215.2023.10179304

[118] Piergiorgio Ladisa, Henrik Plate, Matias Martinez, Olivier Barais, and Serena Elisa Ponta. 2022. Towards the detection of malicious java packages. In *Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*, 63–72.

[119] Piergiorgio Ladisa, Serena Elisa Ponta, Nicola Ronzoni, Matias Martinez, and Olivier Barais. 2023. On the feasibility of cross-language detection of malicious packages in npm and PyPi. In *Proceedings of the 39th Annual Computer Security Applications Conference*, 71–82.

[120] Piergiorgio Ladisa, Merve Sahin, Serena Elisa Ponta, Marco Rosa, Matias Martinez, and Olivier Barais. 2023. The Hitchhiker's guide to malicious third-party dependencies. In *Proceedings of the 2023 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*, 65–74.

[121] Enrique Larios Vargas, Maurício Aniche, Christoph Treude, Magiel Bruntink, and Georgios Gousios. 2020. Selecting third-party libraries: The practitioners' perspective. In *28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 245–256.

[122] Junjie Li, Fazle Rabbi, Cheng Cheng, Aseem Sangalay, Yuan Tian, and Jinqiu Yang. 2024. An exploratory study on fine-tuning large language models for secure code generation. arXiv:2408.09078. Retrieved from https://arxiv.org/abs/2408.09078

[123] Kaixuan Li, Jian Zhang, Sen Chen, Han Liu, Yang Liu, and Yixiang Chen. 2024. PatchFinder: A two-phase approach to security patch tracing for disclosed vulnerabilities in open-source software. In *33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 590–602.

[124] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. StarCoder: May the source be with you! arXiv:2305.06161. Retrieved from https://arxiv.org/abs/2305.06161

[125] Ziyang Li, Saikat Dutta, and Mayur Naik. 2024. LLM-assisted static analysis for detecting security vulnerabilities. arXiv:2405.17238. Retrieved from https://arxiv.org/abs/2405.17238

[126] Zhi Li, Weijie Liu, Hongbo Chen, XiaoFeng Wang, Xiaojing Liao, Luyi Xing, Mingming Zha, Hai Jin, and Deqing Zou. 2022. Robbery on DevOps: Understanding and mitigating illicit cryptomining on continuous integration service platforms. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2397–2412.

[127] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. VulDeePecker: A deep learning-based system for vulnerability detection. arXiv:1801.01681. Retrieved from https://arxiv.org/abs/1801.01681

[128] Linux Foundation. 2022. The State of Software Bill of Materials (SBOM) and Cybersecurity Readiness. Retrieved from https://www.linuxfoundation.org/research/the-state-of-software-bill-of-materials-sbom-and-cybersecurity-readiness

[129] Chengwei Liu, Sen Chen, Lingling Fan, Bihuan Chen, Yang Liu, and Xin Peng. 2022. Demystifying the vulnerability propagation and its evolution via dependency trees in the npm ecosystem. In *44th International Conference on Software Engineering*, 672–684.

[130] Xin Liu, Yixiong Wu, Qingchen Yu, Shangru Song, Yue Liu, Qingguo Zhou, and Jianwei Zhuge. 2022. PG-VulNet: Detect supply chain vulnerabilities in IoT devices using pseudo-code and graphs. In *16th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 205–215.

[131] Yi Liu, Gelei Deng, Yuekang Li, Kailong Wang, Zihao Wang, Xiaofeng Wang, Tianwei Zhang, Yepang Liu, Haoyu Wang, Yan Zheng, et al. 2023. Prompt injection attack against LLM-integrated applications. arXiv:2306.05499. Retrieved from https://arxiv.org/abs/2306.05499

[132] Zhihong Liu, Qing Liao, Wenchao Gu, and Cuiyun Gao. 2023. Software vulnerability detection with GPT and in-context learning. In *2023 8th International Conference on Data Science in Cyberspace (DSC)*. IEEE, 229–236.

[133] Guilong Lu, Xiaolin Ju, Xiang Chen, Wenlong Pei, and Zhilong Cai. 2024. GRACE: Empowering LLM-based software vulnerability detection with graph structure and in-context learning. *Journal of Systems and Software* 212 (2024), 112031.

[134] Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung-won Hwang, and Alexey Svyatkovskiy. 2022. ReACC: A retrieval-augmented code completion framework. arXiv:2203.07722. Retrieved from https://arxiv.org/abs/2203.07722

[135] Fabio Massacci and Viet Hung Nguyen. 2010. Which is the right source for vulnerability studies? An empirical analysis on Mozilla Firefox. In *6th International Workshop on Security Measurements and Metrics*, 1–8.

[136] Michael Meli, Matthew R. McNiece, and Bradley Reaves. 2019. How bad can it Git? Characterizing secret leakage in public GitHub repositories. In *Network and Distributed Systems Security (NDSS) Symposium*.

[137] Microsoft. 2019. OSS Detect Backdoor. Retrieved September 25, 2024 from https://github.com/microsoft/OSSGadget/wiki/OSS-Detect-Backdoor

[138] Courtney Miller, Mahmoud Jahanshahi, Audris Mockus, Bogdan Vasilescu, and Christian Kästner. 2025. Understanding the response to open-source dependency abandonment in the npm ecosystem. In *International Conference on Software Engineering (ICSE)*.

[139] Courtney Miller, Christian Kästner, and Bogdan Vasilescu. 2023. "We feel like we're winging it:" A study on navigating open-source dependency abandonment. In *Proceedings of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM Press, New York, NY.

[140] Hamid Mohayeji, Andrei Agaronian, Eleni Constantinou, Nicola Zannone, and Alexander Serebrenik. 2023. Investigating the resolution of vulnerable dependencies with dependabot security updates. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. IEEE, 234–246.

[141] Ahmad Mohsin, Helge Janicke, Adrian Wood, Iqbal H Sarker, Leandros Maglaras, and Naeem Janjua. 2024. Can we trust large language models generated code? A framework for in-context learning, security patterns, and code evaluations across diverse LLMs. arXiv:2406.12513. Retrieved from https://arxiv.org/abs/2406.12513

[142] Marina Moore, Trishank Karthik Kuppusamy, and Justin Cappos. 2023. Artemis: Defanging software supply chain attacks in multi-repository update systems. In *39th Annual Computer Security Applications Conference (ACSAC '23)*. ACM, New York, NY, 83–97. DOI: https://doi.org/10.1145/3627106.3627129

[143] Suhaib Mujahid, Diego Elias Costa, Rabe Abdalkareem, Emad Shihab, Mohamed Aymen Saied, and Bram Adams. 2021. Toward using package centrality trend to identify packages in decline. *IEEE Transactions on Engineering Management* 69, 6 (2021), 3618–3632.
[144] Siddharth Muralee, Igibek Koishybayev, Aleksandr Nahapetyan, Greg Tystahl, Brad Reaves, Antonio Bianchi, William Enck, Alexandros Kapravelos, and Aravind Machiry. 2023. ARGUS: A framework for staged static taint analysis of GitHub workflows and actions. In *USENIX Security Symposium*.
[145] Zachary Newman, John Speed Meyers, and Santiago Torres-Arias. 2022. Sigstore: Software signing for everybody. In *2022 ACM SIGSAC Conference on Computer and Communications Security*, 2353–2367.
[146] Son Nguyen, Thanh Trong Vu, and Hieu Dinh Vo. 2023. VFFINDER: A graph-based approach for automated silent vulnerability-fix identification. In *2023 15th International Conference on Knowledge and Systems Engineering (KSE)*. IEEE, 1–6.
[147] Truong Giang Nguyen, Thanh Le-Cong, Hong Jin Kang, Xuan-Bach D Le, and David Lo. 2022. VulCurator: A vulnerability-fixing commit detector. In *30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 1726–1730.
[148] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. CodeGen: An open large language model for code with multi-turn program synthesis. arXiv:2203.13474. Retrieved from https://arxiv.org/abs/2203.13474
[149] Liang-bo Ning, Shijie Wang, Wenqi Fan, Qing Li, Xin Xu, Hao Chen, and Feiran Huang. 2024. CheatAgent: Attacking LLM-empowered recommender systems via LLM agent. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2284–2295.
[150] NIST. 2022. NIST Special Publication 800-218 Secure Software Development Framework (SSDF). Retrieved from https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-218.pdf
[151] NIST. 2022. NIST Special Publication 800-161 Rev 1 Cybersecurity Supply Chain Risk Management Practices for Systems and Organizations. Retrieved May 2022 from https://csrc.nist.gov/pubs/sp/800/161/r1/final
[152] NTIA. 2021. The Minimal Elements of a Software Bill of Materials. Retrieved July 21, 2021 from https://www.ntia.doc.gov/files/ntia/publications/sbom_minimum_elements_report.pdf
[153] National Institute of Standards and Technology (NIST). 2024. National Vulnerability Database (NVD). Retrieved September 25, 2024 from https://nvd.nist.gov/
[154] Marc Ohm, Felix Boes, Christian Bungartz, and Michael Meier. 2022. On the feasibility of supervised machine learning for the detection of malicious software packages. In *17th International Conference on Availability, Reliability and Security*, 1–10.
[155] Marc Ohm, Lukas Kempf, Felix Boes, and Michael Meier. 2022. *Towards Detection of Malicious Software Packages Through Code Reuse by Malevolent Actors*. Gesellschaft für Informatik, Bonn.
[156] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. 2020. Backstabber's knife collection: A review of open source software supply chain attacks. In *17th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA '20)*. Springer, 23–43.
[157] Theo X Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. 2023. Is self-repair a silver bullet for code generation? In *12th International Conference on Learning Representations*.
[158] Marwan Omar. 2023. Detecting software vulnerabilities using language models. arXiv:2302.11773. Retrieved from https://arxiv.org/abs/2302.11773
[159] Hassan Onsori Delicheh, Alexandre Decan, and Tom Mens. 2024. Quantifying security issues in reusable JavaScript actions in GitHub workflows. In *21st International Conference on Mining Software Repositories*, 692–703.
[160] OpenSSF. 2023. SLSA Tech Talk Highlights. Retrieved from https://openssf.org/blog/2023/10/20/slsa-tech-talk-highlights/
[161] OpenSSF. 2023. Supply-chain Levels for Software Artifacts (SLSA). Retrieved from https://slsa.dev/
[162] OpenSSF. 2023. Secure Supply Chain Consumption Framework (S2C2F). Retrieved July 2023 from https://github.com/ossf/s2c2f
[163] OpenSSF. 2024. Repository Service for TUF (RSTUF). Retrieved September 26, 2024 from https://openssf.org/projects/repository-service-for-tuf/
[164] OWASP. 2024. Software Component Verification Standard. Retrieved from https://scvs.owasp.org/
[165] Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. 2018. Vulnerable open source dependencies: Counting those that matter. In *12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 1–10.
[166] Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. 2020. Vuln4Real: A methodology for counting actually vulnerable dependencies. *IEEE Transactions on Software Engineering* 48, 5 (2020), 1592–1609.

[167] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the keyboard? Assessing the security of GitHub copilot's code contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 754–768.
[168] Henrik Plate. 2023. SBOM vs. SBOM: Comparing SBOMs from different tools and lifecycle stages. Retrieved from https://www.endorlabs.com/learn/sbom-vs-sbom-comparing-sboms-from-different-tools-and-lifecycle-stages
[169] Serena Elisa Ponta, Henrik Plate, and Antonino Sabetta. 2018. Beyond metadata: Code-centric and usage-based analysis of known vulnerabilities in open-source software. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 449–460.
[170] Serena Elisa Ponta, Henrik Plate, and Antonino Sabetta. 2020. Detection, assessment and mitigation of vulnerabilities in open source dependencies. *Empirical Software Engineering* 25, 5 (2020), 3175–3215.
[171] Gede Artha Azriadi Prana, Abhishek Sharma, Lwin Khin Shar, Darius Foo, Andrew E. Santosa, Asankhaya Sharma, and David Lo. 2021. Out of sight, out of mind? How vulnerable dependencies affect open-source projects. *Empirical Software Engineering* 26 (2021), 1–34.
[172] Moumita Das Purba, Arpita Ghosh, Benjamin J. Radford, and Bill Chu. 2023. Software vulnerability detection using large language models. In *2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 112–119.
[173] PyPA. 2021. Python Packaging Advisory Database. Retrieved from https://github.com/pypa/advisory-database
[174] pyup.io. 2024. Safety DB. Retrieved from https://github.com/pyupio/safety-db
[175] Md Fazle Rabbi, Arifa Islam Champa, Costain Nachuma, and Minhaz Fahim Zibran. 2024. Sbom generation tools under microscope: A focus on the npm ecosystem. In *39th ACM/SIGAPP Symposium on Applied Computing*, 1233–1241.
[176] Md Fazle Rabbi, Arifa Islam Champa, Costain Nachuma, and Minhaz Fahim Zibran. 2024. SBOM generation tools under microscope: A focus on the npm ecosystem. In *39th ACM/SIGAPP Symposium on Applied Computing*. ACM, New York, NY.
[177] Imranur Rahman, Ranidya Paramitha, Henrik Plate, Dominik Wermke, and Laurie Williams. 2024. Less is more: A mixed-methods study on security-sensitive API calls in Java for better dependency selection. arXiv:2408.02846. Retrieved from https://arxiv.org/abs/2408.02846
[178] Imranur Rahman, Nusrat Zahan, Stephen Magill, William Enck, and Laurie Williams. 2024. Characterizing dependency update practice of NPM, PyPI and Cargo packages. arXiv:2403.17382. Retrieved from https://arxiv.org/abs/2403.17382
[179] Harshini Sri Ramulu, Helen Schmitt, Dominik Wermke, and Yasemin Acar. 2024. Security and privacy software creators' perspectives on unintended consequences. In *Proceedings of the 33rd USENIX Security Symposium (USENIX Sec '24)*. USENIX Association. Retrieved August 9, 2023 from https://www.usenix.org/conference/usenixsecurity23/presentation/krause
[180] David Reid, Kristiina Rahkema, and James Walden. 2023. Large scale study of orphan vulnerabilities in the software supply chain. In *19th International Conference on Predictive Models and Data Analytics in Software Engineering*, 22–32.
[181] Zhilei Ren, He Jiang, Jifeng Xuan, and Zijiang Yang. 2018. Automated localization for unreproducible builds. In *40th International Conference on Software Engineering (ICSE '18)*. ACM, New York, NY, 71–81. DOI: https://doi.org/10.1145/3180155.3180224
[182] Zhilei Ren, Changlin Liu, Xusheng Xiao, He Jiang, and Tao Xie. 2019. Root cause localization for unreproducible builds via causality analysis over system call tracing. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE '19)*, 527–538. DOI: https://doi.org/10.1109/ASE.2019.00056
[183] Zhilei Ren, Shiwei Sun, Jifeng Xuan, Xiaochen Li, Zhide Zhou, and He Jiang. 2022. Automated patching for unreproducible builds. In *44th International Conference on Software Engineering (ICSE '22)*. ACM, New York, NY, 200–211. DOI: https://doi.org/10.1145/3510003.3510102
[184] Renovatebot. 2024. Renovatebot: Automating Dependency Updates. Retrieved September 25, 2024 from https://github.com/renovatebot/renovate
[185] Atanas Rountev, Scott Kagan, and Michael Gibas. 2004. Static and dynamic analysis of call chains in Java. In *2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, 1–11.
[186] RustSec. 2018. RustSec Advisory Database. Retrieved from https://github.com/rustsec/advisory-db
[187] Antonino Sabetta, Serena Elisa Ponta, Rocio Cabrera Lozoya, Michele Bezzi, Tommaso Sacchetti, Matteo Greco, Gergő Balogh, Péter Hegedűs, Rudolf Ferenc, Ranindya Paramitha, et al. 2024. Known vulnerabilities of open source projects: Where are the fixes? *IEEE Security & Privacy* 22 (2024), 49–59.
[188] Justin Samuel, Nick Mathewson, Justin Cappos, and Roger Dingledine. 2010. Survivable key compromise in software update systems. In *17th ACM Conference on Computer and Communications Security*, 61–72.
[189] Natalie M. Scala, Allison C. Reilly, Paul L. Goethals, and Michel Cukier. 2019. Risk and the five hard problems of cybersecurity. *Risk Analysis* 39, 10 (2019), 2119–2126.

[190] Simone Scalco and Ranindya Paramitha. 2024. Hash4Patch: A lightweight low false positive tool for finding vulnerability patch commits. In *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*. IEEE, 733–737.
[191] Simone Scalco, Ranindya Paramitha, Duc-Ly Vu, and Fabio Massacci. 2022. On the feasibility of detecting injections in malicious npm packages. In *Proceedings of the 17th International Conference on Availability, Reliability and Security*, 1–8.
[192] Scorecard. 2021. Security Scorecards for Open Source Projects. Retrieved from https://github.com/ossf/scorecard
[193] Adriana Sejfia and Max Schäfer. 2022. Practical automated detection of malicious npm packages. In *44th International Conference on Software Engineering*, 1681–1692.
[194] Aman Sharma, Martin Wittlinger, Benoit Baudry, and Martin Monperrus. 2024. SBOM.EXE: Countering dynamic code injection based on software bill of materials in Java. arXiv:2407.00246. Retrieved from https://arxiv.org/abs/2407.00246
[195] Xinyue Shen, Zeyuan Chen, Michael Backes, Yun Shen, and Yang Zhang. 2023. "Do anything now": Characterizing and evaluating in-the-wild jailbreak prompts on large language models. arXiv:2308.03825. Retrieved from https://arxiv.org/abs/arXiv:2308.03825
[196] Rajulapati Shourya, Yoko Kumagai, C. Ashokkumar, Hiroki Yamazaki, and Hirofumi Nakakoji. 2023. Proposal of vulnerability assessment tool for software supply chain security. *Journal of Information Processing* 31 (2023), 842–850.
[197] Ilia Shumailov, Zakhar Shumaylov, Yiren Zhao, Yarin Gal, Nicolas Papernot, and Ross Anderson. 2023. The curse of recursion: Training on generated data makes models forget. arXiv:2305.17493. Retrieved from https://arxiv.org/abs/2305.17493
[198] Sigstore Project. Fulcio Certificate Authority Overview. Sigstore Documentation. Retrieved September 26, 2024 from https://docs.sigstore.dev/certificate_authority/overview/
[199] Sigstore Project. 2024. OIDC Usage in Fulcio. Sigstore *Documentation*. Retrieved September 26, 2024 from https://docs.sigstore.dev/certificate_authority/oidc-in-fulcio/
[200] Sigstore Project. Rekor Logging Overview. Sigstore Documentation. Retrieved September 26, 2024 from https://docs.sigstore.dev/logging/overview/
[201] Tanmay Singla, Dharun Anandayuvaraj, Kelechi G. Kalu, Taylor R. Schorlemmer, and James C. Davis. 2023. An empirical study on using large language models to analyze software supply chain security failures. In *2023 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*, 5–15.
[202] SLSA. 2022. SBOM + SLSA: Accelerating SBOM success with the help of SLSA. Retrieved from https://slsa.dev/blog/2022/05/slsa-sbom
[203] Snyk. 2024. Snyk vulnerability database. Retrieved from https://snyk.io/vuln
[204] SolarWinds. 2021. SolarWinds Security Advisory. Retrieved April 6, 2021 from https://www.solarwinds.com/sa-overview/securityadvisory
[205] Sonatype. 2021. 7th State of the Software Supply Chain. Retrieved from https://www.sonatype.com/resources/state-of-the-software-supply-chain-2021
[206] Sonatype. 2024. State of the Software Supply Chain: A Decade of Data. Retrieved from https://www.sonatype.com/en/press-releases/sonatypes-10th-annual-state-of-the-software-supply-chain-report
[207] Joseph Spracklen, Raveen Wijewickrama, A. H. M. Sakib, Anindya Maiti, and Murtuza Jadliwala. 2024. We have a package for you! A comprehensive analysis of package hallucinations by code generating LLMs. arXiv:2406.10279. Retrieved from https://arxiv.org/abs/2406.10279
[208] Trevor Stalnaker, Nathan Wintersgill, Oscar Chaparro, Massimiliano Di Penta, Daniel M. German, and Denys Poshyvanyk. 2024. Boms away! Inside the minds of stakeholders: A comprehensive study of bills of materials for software systems. In *46th IEEE/ACM International Conference on Software Engineering*, 1–13.
[209] Igor Steinmacher, Marco Aurelio Graciotto Silva, Marco Aurelio Gerosa, and David F. Redmiles. 2015. A systematic literature review on the barriers faced by newcomers to open source software projects. *Information and Software Technology* 59 (2015), 67–85.
[210] Jacob Stringer, Amjed Tahir, Kelly Blincoe, and Jens Dietrich. 2020. Technical lag of dependencies in major package managers. In *2020 27th Asia-Pacific Software Engineering Conference (APSEC)*, 228–237. DOI: https://doi.org/10.1109/APSEC51365.2020.00031
[211] Donald Stufft, Justin Cappos, and Trishank Karthik Kuppusamy. 2014. PEP 458—Secure PyPI Downloads with TUF. Retrieved September 26, 2024 from https://peps.python.org/pep-0458/#pypi-and-tuf-metadata
[212] Jiamou Sun, Zhenchang Xing, Qinghua Lu, Xiwei Xu, Liming Zhu, Thong Hoang, and Dehai Zhao. 2023. Silent vulnerable dependency alert prediction with vulnerability key aspect explanation. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 970–982.
[213] Zhi Sun, Zhaoheng Quan, Shangren Yu, Ling Zhang, and Dengming Mao. 2024. A knowledge-driven framework for software supply chain security analysis. In *2024 8th International Conference on Control Engineering and Artificial Intelligence*, 267–272.

[214] Synopsys. 2023. Open Source Security and Risk Analysis (OSSRA). Retrieved from https://www.synopsys.com/software-integrity/resources/analyst-reports/open-source-security-risk-analysis.html

[215] Synopsys 2024. 2024 Open Source Security and Risk Analysis (OSSRA) Report. Technical Report, CA, USA. Retrieved from https://www.synopsys.com/software-integrity/resources/analyst-reports/open-source-security-risk-analysis.html

[216] Marc Szafraniec, Baptiste Roziere, Hugh Leather, Francois Charton, Patrick Labatut, and Gabriel Synnaeve. 2022. Code translation with compiler representations. arXiv:2207.03578. Retrieved from https://arxiv.org/abs/2207.03578

[217] Mahzabin Tamanna, Sivana Hamer, Mindy Tran, Sascha Fahl, Yasemin Acar, and Laurie Williams. 2024. Unraveling challenges with supply-chain levels for software artifacts (SLSA) for securing the software supply chain. arXiv:2409.05014. Retrieved from https://arxiv.org/abs/2409.05014

[218] Xin Tan, Yuan Zhang, Chenyuan Mi, Jiajun Cao, Kun Sun, Yifan Lin, and Min Yang. 2021. Locating the security patches for disclosed OSS vulnerabilities with vulnerability-commit correlation ranking. In *2021 ACM SIGSAC Conference on Computer and Communications Security*, 3282–3299.

[219] Minaoar Hossain Tanzil, Gias Uddin, and Ann Barcomb. 2024. "How do people decide?": A model for software library selection. In *2024 IEEE/ACM 17th International Conference on Cooperative and Human Aspects of Software Engineering*, 1–12.

[220] Matthew Taylor, Ruturaj K Vaidya, Drew Davidson, Lorenzo De Carli, and Vaibhav Rastogi. 2020. SpellBound: Defending against package typosquatting. arXiv:2003.03471. Retrieved from https://arxiv.org/abs/2003.03471

[221] Chandra Thapa, Seung Ick Jang, Muhammad Ejaz Ahmed, Seyit Camtepe, Josef Pieprzyk, and Surya Nepal. 2022. Transformer-based language models for software vulnerability detection. In *38th Annual Computer Security Applications Conference*, 481–496.

[222] The White House. 2021. Executive Order on America's Supply Chains (EO14017). Retrieved from https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity/

[223] Ken Thompson. 1984. Reflections on trusting trust. *Communications of the ACM* 27, 8 (Aug. 1984), 761–763. DOI: https://doi.org/10.1145/358198.358210

[224] Tidelift. 2024. The 2024 Tidelift State of the Open Source Maintainer Report. Retrieved from https://explore.tidelift.com/2024-survey

[225] Santiago Torres-Arias, Hammad Afzali, Trishank Karthik Kuppusamy, Reza Curtmola, and Justin Cappos. 2019. In-toto: Providing farm-to-table guarantees for bits and bytes. In *28th USENIX Security Symposium (USENIX Security '19)*, 1393–1410.

[226] Santiago Torres-Arias, Dan Geer, and John Speed Meyers. 2023. A viewpoint on knowing software: Bill of materials quality when you see it. *IEEE Security & Privacy* 21, 6 (Nov. 2023), 50–54.

[227] Mindy Tran, Yasemin Acar, Michel Cucker, William Enck, Alexandros Kapravelos, Christian Kastner, and Laurie Williams. 2022. S3C2 summit 2022-09: Industry secure supply chain summit. arXiv:2307.15642. Retrieved from http://arxiv.org/abs/2307.15642

[228] Greg Tystahl, Yasemin Acar, Michel Cucker, William Enck, Alexandros Kapravelos, Christian Kastner, and Laurie Williams. 2024. S3C2 summit 2024-03: Industry secure supply chain summit. arXiv: 2405.08762. Retrieved from https://arxiv.org/abs/2405.08762

[229] Greg Tystahl, Yasemin Acar, Michel Cukier, William Enck, Christian Kastner, Alexandros Kapravelos, Dominik Wermke, and Laurie Williams. 2024. S3C2 summit 2024-03: Industry secure supply chain summit. arXiv:2405.08762. Retrieved from https://arxiv.org/abs/2405.08762

[230] Bibek Upadhayay and Vahid Behzadan. 2024. Sandwich attack: Multi-language mixture adaptive attack on LLMs. arXiv:2404.07242. Retrieved from https://arxiv.org/abs/2404.07242

[231] USENIX Security. 2024. USENIX Security Ethics Guidelines. Retrieved from https://www.usenix.org/conference/usenixsecurity25/ethics-guidelines

[232] Pablo Valenzuela-Toledo and Alexandre Bergel. 2022. Evolution of GitHub action workflows. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 123–127.

[233] Marat Valiev, Bogdan Vasilescu, and James Herbsleb. 2018. Ecosystem-level determinants of sustained activity in open-source projects: A case study of the PyPI ecosystem. In *2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 644–655.

[234] Nikos Vasilakis, Achilles Benetopoulos, Shivam Handa, Alizee Schoen, Jiasi Shen, and Martin C. Rinard. 2021. Supply-chain vulnerability elimination via active learning and regeneration. In *2021 ACM SIGSAC Conference on Computer and Communications Security*, 1755–1770.

[235] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, and Jonathan M. Smith. 2018. BreakApp: Automated, flexible application compartmentalization. In *NDSS Network and Distributed Systems Security (NDSS) Symposium*.

[236] Duc-Ly Vu. 2020. A fork of bandit tool with patterns to identifying malicious Python code. Retrieved September 25, 2024 from https://github.com/lyvd/bandit4mal

[237] Duc-Ly Vu, Fabio Massacci, Ivan Pashchenko, Henrik Plate, and Antonino Sabetta. 2021. LastPyMile: Identifying the discrepancy between sources and packages. In *29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 780–792.

[238] Duc-Ly Vu, Zachary Newman, and John Speed Meyers. 2023. Bad snakes: Understanding and improving Python package index malware scanning. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 499–511.

[239] Duc Ly Vu, Ivan Pashchenko, Fabio Massacci, Henrik Plate, and Antonino Sabetta. 2020. Towards using source code repositories to identify software supply chain attacks. In *2020 ACM SIGSAC Conference on Computer and Communications Security*, 2093–2095.

[240] VulnDB. 2018. The Go Vulnerability Database. Retrieved fromhttps://github.com/golang/vulndb

[241] Shenao Wang, Yanjie Zhao, Xinyi Hou, and Haoyu Wang. 2024. Large language model supply chain: A research agenda. arXiv:2404.12736. Retrieved from https://arxiv.org/abs/2404.12736

[242] Ying Wang, Peng Sun, Lin Pei, Yue Yu, Chang Xu, Shing-Chi Cheung, Hai Yu, and Zhiliang Zhu. 2023. Plumber: Boosting the propagation of vulnerability fixes in the npm ecosystem. *IEEE Transactions on Software Engineering* 49, 5 (2023), 3155–3181.

[243] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. arXiv:2109.00859. Retrieved from https://arxiv.org/abs/2109.00859

[244] Dominik Wermke, Jan H. Klemmer, Noah Wöhler, Juliane Schmüser, Harshini SriRamulu, YaseminAcar, and SaschaFahl. 2023. "Always contribute back": A qualitative study on security challenges of the open source supply Chain. In *44th IEEE Symposium on Security and Privacy (IEEE S&P '23)*. IEEE. Retrieved May 22, 2023 fromhttps://www.ieee-security.org/TC/SP2023/program-papers.html

[245] Dominik Wermke, Noah Wöhler, Jan H. Klemmer, Marcel Fourné, Yasemin Acar, and Sascha Fahl. 2022. Committed to trust: A qualitative study on security & trust in open source software projects. In *43rd IEEE Symposium on Security and Privacy*. Retrieved May 22, 2024 from https://www.ieee-security.org/TC/SP2022/index.html

[246] Laurie Williams, Sammy Migues, Jamie Boote, and Ben Hutchison. 2024. Proactive software supply chain risk management framework (P-SSCRM) version 1. arXiv:2404.12300. Retrieved from https://arxiv.org/abs/2404.12300

[247] Susheng Wu, Wenyan Song, Kaifeng Huang, Bihuan Chen, and Xin Peng. 2024. Identifying affected libraries and their ecosystems for open source software vulnerabilities. In *IEEE/ACM 46th International Conference on Software Engineering*, 1–12.

[248] Susheng Wu, Ruisi Wang, Kaifeng Huang, Yiheng Cao, Wenyan Song, Zhuotong Zhou, Yiheng Huang, Bihuan Chen, and Xin Peng. 2024. Vision: Identifying affected library versions for open source software vulnerabilities. In *39th IEEE/ACM International Conference on Automated Software Engineering*, 1447–1459.

[249] Boming Xia, Dawen Zhang, Yue Liu, Qinghua Lu, Zhenchang Xing, and Liming Zhu. 2024. Trust in software supply chains: Blockchain-enabled SBOM and the AIBOM future. In *2024 ACM/IEEE 4th International Workshop on Engineering and Cybersecurity of Critical Systems (EnCyCriS) and 2024 IEEE/ACM Second International Workshop on Software Vulnerability*. ACM, New York, NY, 12–19.

[250] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2022. Practical program repair in the era of large pre-trained language models. arXiv:2210.14179. Retrieved from https://arxiv.org/abs/2210.14179

[251] Congying Xu, Bihuan Chen, Chenhao Lu, Kaifeng Huang, Xin Peng, and Yang Liu. 2022. Tracking patches for open source software vulnerabilities. In *30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22)*. ACM, New York, NY, 860–871. DOI: https://doi.org/10.1145/3540250.3549125

[252] Jiacen Xu, Jack W. Stokes, Geoff McDonald, Xuesong Bai, David Marshall, Siyue Wang, Adith Swaminathan, and Zhou Li. 2024. AutoAttacker: A large language model guided system to implement automatic cyber-attacks. arXiv:2403.01038. Retrieved from https://arxiv.org/abs/2403.01038

[253] Zhihao Xu, Ruixuan Huang, Xiting Wang, Fangzhao Wu, Jing Yao, and Xing Xie. 2024. Uncovering safety risks in open-source LLMs through concept activation vector. arXiv:2404.12038. Retrieved from https://arxiv.org/abs/2404.12038

[254] Dapeng Yan, Yuqing Niu, Kui Liu, Zhe Liu, Zhiming Liu, and Tegawendé F. Bissyandé. 2021. Estimating the attack surface from residual vulnerabilities in open source software supply chain. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 493–502.

[255] Zhou Yang, Zhensu Sun, Terry Zhuo Yue, Premkumar Devanbu, and David Lo. 2024. Robustness, security, privacy, explainability, efficiency, and usability of large language models for code. arXiv:2403.07506. Retrieved from https://arxiv.org/abs/2403.07506

[256] Aditya Sirish A. Yelgundhalli and Justin Cappos. 2024. Introducing gittuf: A security layer for Git repositories. Retrieved September 26, 2024 from https://openssf.org/blog/2024/01/18/introducing-gittuf-a-security-layer-for-git-repositories/

[257] Sangyeop Yeo, Yu-Seung Ma, Sang Cheol Kim, Hyungkook Jun, and Taeho Kim. 2024. Framework for evaluating code generation ability of large language models. *ETRI Journal* 46, 1 (2024), 106–117.

[258] Awad A. Younis, Yashwant K. Malaiya, and Indrajit Ray. 2014. Using attack surface entry points and reachability analysis to assess the risk of software vulnerability exploitability. In *2014 IEEE 15th International Symposium on High-Assurance Systems Engineering*. IEEE, 1–8.

[259] Sheng Yu, Wei Song, Xunchao Hu, and Heng Yin. 2024. On the correctness of metadata-based SBOM generation: A differential analysis approach. In *2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 29–36.

[260] Zeliang Yu, Ming Wen, Xiaochen Guo, and Hai Jin. 2024. Maltracker: A fine-grained NPM malware tracker copiloted by LLM-enhanced dataset. In *33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 1759–1771.

[261] Nusrat Zahan, Yasemin Acar, Michel Cucker, William Enck, Alexandros Kapravelos, Christian Kastner, and Laurie Williams. 2023. S3C2 summit 2023-11: Industry secure supply chain summit. arXiv:2408.16529. Retrieved from https://arxiv.org/abs/2408.16529

[262] Nusrat Zahan, Philipp Burckhardt, Mikola Lysenko, Feross Aboukhadijeh, and Laurie Williams. 2024. MalwareBench: Malware samples are not enough. In *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*. IEEE, 728–732.

[263] Nusrat Zahan, Philipp Burckhardt, Mikola Lysenko, Feross Aboukhadijeh, and Laurie Williams. 2024. Shifting the lens: Detecting malware in npm ecosystem with large language models. arXiv:2403.12196. Retrieved from https://arxiv.org/abs/2403.12196

[264] Nusrat Zahan, Parth Kanakiya, Brian Hambleton, Shohanuzzaman Shohan, and Laurie Williams. 2023. OpenSSF scorecard: On the path toward ecosystem-wide automated security metrics. *IEEE Security & Privacy* 21, 6 (2023), 76–88. DOI: https://doi.org/10.1109/MSEC.2023.3279773

[265] Nusrat Zahan, Elizabeth Lin, Mahzabin Tamanna, William Enck, and Laurie Williams. 2023. Software bills of materials are required. are we there yet? *IEEE Security & Privacy* 21, 2 (2023), 82–88. DOI: https://doi.org/10.1109/MSEC.2023.3237100

[266] Nusrat Zahan, Shohanuzzaman Shohan, Dan Harris, and Laurie Williams. 2023. Do software security practices yield fewer vulnerabilities? In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 292–303.

[267] Nusrat Zahan, Thomas Zimmermann, Patrice Godefroid, Brendan Murphy, Chandra Maddila, and Laurie Williams. 2022. What are weak links in the npm supply chain? In *44th International Conference on Software Engineering: Software Engineering in Practice*, 331–340.

[268] Ahmed Zerouali, Eleni Constantinou, Tom Mens, Gregorio Robles, and Jesus Gonzalez-Barahona. 2018. An empirical analysis of technical lag in npm package dependencies. In *17th International Conference on Software and Software Reuse (ICSR)*. DOI: https://doi.org/10.1007/978-3-319-90421-4_6

[269] Ahmed Zerouali, Tom Mens, Alexandre Decan, and Coen De Roover. 2022. On the impact of security vulnerabilities in the npm and RubyGems dependency networks. *Empirical Software Engineering* 27, 5 (2022), 107.

[270] Chenyuan Zhang, Hao Liu, Jiutian Zeng, Kejing Yang, Yuhong Li, and Hui Li. 2024. Prompt-enhanced software vulnerability detection using ChatGPT. In *2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, 276–277.

[271] Fangyuan Zhang, Lingling Fan, Sen Chen, Miaoying Cai, Sihan Xu, and Lida Zhao. 2024. Does the vulnerability threaten our projects? Automated vulnerable API detection for third-party libraries. *IEEE Transactions on Software Engineering* (2024).

[272] Junan Zhang, Kaifeng Huang, Bihuan Chen, Chong Wang, Zhenhao Tian, and Xin Peng. 2023. Malicious package detection in NPM and PyPI using a single model of malicious behavior sequence. arXiv:2309.02637. Retrieved from https://arxiv.org/abs/2309.02637

[273] Lyuye Zhang, Chengwei Liu, Sen Chen, Zhengzi Xu, Lingling Fan, Lida Zhao, Yiran Zhang, and Yang Liu. 2023. Mitigating persistence of open-source vulnerabilities in maven ecosystem. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 191–203.

[274] Lyuye Zhang, Chengwei Liu, Zhengzi Xu, Sen Chen, Lingling Fan, Lida Zhao, Jiahui Wu, and Yang Liu. 2023. Compatible remediation on vulnerabilities from third-party libraries for Java projects. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2540–2552.

[275] Yue Zhang, Yafu Li, Leyang Cui, Deng Cai, Lemao Liu, Tingchen Fu, Xinting Huang, Enbo Zhao, Yu Zhang, Yulong Chen, et al. 2023. Siren's song in the AI ocean: A survey on hallucination in large language models. arXiv:2309.01219. Retrieved from https://arxiv.org/abs/2309.01219

[276] Ying Zhang, Wenjia Song, Zhengjie Ji, Danfeng (Daphne) Yao, and Na Meng. 2023. How well does LLM generate security tests? arXiv:2310.00710. Retrieved from https://arxiv.org/abs/2310.00710

[277] Ying Zhang, Xiaoyan Zhou, Hui Wen, Wenjia Niu, Jiqiang Liu, Haining Wang, and Qiang Li. 2024. Tactics, techniques, and procedures (TTPs) in interpreted malware: A zero-shot generation with large language models. arXiv:2407.08532. Retrieved from https://arxiv.org/abs/2407.08532

[278] Wanru Zhao, Vidit Khazanchi, Haodi Xing, Xuanli He, Qiongkai Xu, and Nicholas Donald Lane. 2024. Attacks on third-party APIs of large language models. arXiv:2404.16891. Retrieved from https://arxiv.org/abs/2404.16891

[279] Zibin Zheng, Kaiwen Ning, Jiachi Chen, Yanlin Wang, Wenqing Chen, Lianghong Guo, and Weicheng Wang. 2023. Towards an understanding of large language models in software engineering tasks. arXiv:2308.11396. Retrieved from https://arxiv.org/abs/2308.11396

[280] Jiayuan Zhou, Michael Pacheco, Jinfu Chen, Xing Hu, Xin Xia, David Lo, and Ahmed E. Hassan. 2023. Colefunda: Explainable silent vulnerability fix identification. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2565–2577.

[281] Jiayuan Zhou, Michael Pacheco, Zhiyuan Wan, Xin Xia, David Lo, Yuan Wang, and Ahmed E. Hassan. 2021. Finding a needle in a haystack: Automated mining of silent vulnerability fixes. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 705–716.

[282] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *33rd International Conference on Neural Information Processing Systems*, 10197–10207.

Received 29 March 2024; revised 13 December 2024; accepted 18 December 2024