Beyond Compliance: A Large Scale Study on the Completeness and Consistency of the GitHub SBOMs

Kawsar Ahmed Bhuiyan

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of

Master of Applied Science (Software Engineering) at

Concordia University

Montréal, Québec, Canada

July 2026

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Kawsar Ahmed Bhuiyan**

Entitled: **Beyond Compliance: A Large Scale Study on the Completeness and Consistency of the GitHub SBOMs**

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Software Engineering)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

_____ Chair
Dr. Juergen Rilling

_____ Examiner
Dr. Juergen Rilling

_____ Examiner
Dr. Emad Shihab

_____ Supervisor
Dr. Diego Elias Costa

Approved by _____
Joey Paquet, Chair
Department of Computer Science and Software Engineering

_____ 2026

_____
Mourad Debbabi, Dean
Faculty of Engineering and Computer Science

Abstract

Beyond Compliance: A Large Scale Study on the Completeness and Consistency of the GitHub SBOMs

Kawsar Ahmed Bhuiyan

Modern software development relies heavily on open-source components. Reusing components accelerates innovation but increases exposure to supply-chain attacks exploiting known vulnerabilities. Software Bills of Materials (SBOMs) improve software supply chain transparency by enumerating components, their versions, and their provenance. GitHub, the largest open-source development hosting platform, now automatically generates SBOMs for repositories, providing valuable metadata for risk assessment. Yet, it is unclear whether GitHub SBOMs can serve as a reliable source for vulnerability and license analysis, and how incomplete or inconsistent metadata may affect different programming ecosystems.

In this thesis, we present a large-scale analysis of 10,000 GitHub repositories across ten programming language ecosystems, evaluating GitHub SBOMs against three other popular SBOM generators: Syft, Trivy, and the Microsoft SBOM Tool. Our study finds a lack of NTIA compliance in GitHub SBOMs, though core metadata is consistently present. We also find that component version and license information availability is highly dependent on the programming ecosystem. Compared with the other three tools, GitHub yields results similar to the Microsoft SBOM Tool and often outperforms Syft and Trivy in providing version and license information. Finally, we discuss potential shortcomings of GitHub SBOMs, directly related to how each ecosystem manages its dependencies.

Acknowledgments

This thesis marks the end of an important chapter in my academic journey. It would not have been possible without the support, guidance, and encouragement of many people, and I would like to express my sincere gratitude to all of them.

First and foremost, I would like to thank my supervisor, **Dr. Diego Elias Costa**, for his guidance, patience, and constant support throughout my master's degree. His feedback, advice, and encouragement helped me grow as a researcher and gave me confidence to overcome many challenges. I am truly grateful for everything I learned under his supervision.

I would also like to thank everyone in the **REALISE Lab** for creating such a friendly and supportive research environment. I learned a great deal from our discussions, collaborations, and everyday interactions. It was a pleasure to be part of a group where people were always willing to help and share their knowledge.

I am grateful to the **NSERC CREATE SE4AI** program for supporting my graduate studies and providing opportunities to learn, collaborate, and connect with researchers and industry professionals. The program has played an important role in my academic and professional growth.

My thanks also go to the faculty members and staff at Concordia University for providing a great learning environment during my graduate studies.

Finally, I would like to thank my family and friends for their endless love, support, and encouragement. Their belief in me gave me the strength to keep going, especially during difficult times. I would not have reached this milestone without them.

Thank you to everyone who has been part of this journey. I will always be grateful for your support and kindness.

Dedication

To my parents,

This thesis is dedicated to you. Your endless love, sacrifices, encouragement, and unwavering belief in me have been the foundation of everything I have achieved. None of this would have been possible without your support. Thank you for always standing by my side and inspiring me to pursue my dreams.

Contents

List of Figures

List of Tables

Chapter 1

Introduction and Research Statement

1.1 Introduction

The increasing reliance on third-party and open-source software components has become a defining characteristic of contemporary software development [1]. While this trend enables faster innovation and reduces engineering costs, it also increases the risk of exploitation, as attackers may target known vulnerabilities in these components, potentially undermining system integrity, exposing sensitive data, or causing significant operational and economic damage [2–4]. These challenges are intensified by the growing scale and complexity of software dependency networks, which often span thousands of interrelated packages [5, 6]. Within such networks, a flaw in a single, widely embedded dependency can affect a disproportionate number of downstream systems: this was illustrated when a vulnerability in Log4j, a popular logging library for Java programs, prompted emergency mitigation guidance from national cybersecurity agencies, leaving many organizations unable to quickly determine whether their own software depended on the affected component [7].

To address these concerns, the Software Bill of Materials (SBOM) has emerged as a key mechanism for enhancing transparency in the software supply chain. An SBOM is a document that provides a structured inventory of all software components, including open-source and proprietary modules, along with their dependencies and relationships [8, 9]. By making this information transparent, SBOMs allow downstream users to assess potential cybersecurity and licensing risks [10, 11].

Among the many uses of an SBOM, supporting vulnerability identification is one of the most direct: when a vulnerability is disclosed in a component, an SBOM lets consumers quickly identify whether their software is affected by that component and assess the risk it introduces [10]. This also lets consumers apply mitigations independently of the software supplier [10]. Realizing this use case in practice, however, depends on an SBOM accurately and completely recording each component's identity and version, the very metadata whose completeness and consistency this thesis investigates.

The strategic importance of SBOMs was recognized in the U.S. Executive Order on Improving the Nation's Cybersecurity, issued in May 2021, which mandated the use of SBOMs in federal software procurement [12]. To promote standardization, the National Telecommunications and Information Administration (NTIA) established the Software Transparency initiative, defining guidelines for machine-readable SBOMs that specify minimum required elements such as supplier and component names, version information, dependency relationships, and other essential metadata [13]. These guidelines aim to harmonize SBOM practices across organizations, enabling scalable and interoperable adoption.

Within this context, GitHub has emerged as a significant platform for SBOM generation and dissemination. GitHub hosts millions of open-source repositories and provides native support for SBOM creation by leveraging its dependency graph [14]. Hence, GitHub is a compelling target for empirical investigation, particularly given its ecosystem diversity and scale. As users may adopt GitHub SBOMs as part of their compliance and quality assessment [15, 16], it is crucial to assess their compliance and consistency and to compare them with other widely used SBOM generation tools.

In this thesis, we present a large-scale empirical study of GitHub-generated SBOMs across 10 popular programming language ecosystems: C, C++, C#, Python, PHP, Java, JavaScript, Go, Swift, and Rust. We collect SBOM data from 10,000 GitHub repositories, and design our study to answer the following research questions:

RQ1: How compliant and complete are GitHub SBOMs? We examine whether GitHub SBOMs include the key metadata required for software supply chain activities, focusing on the compliance to NTIA minimum elements. We then evaluate the presence of version and license information of GitHub SBOMs' components. We found that GitHub SBOMs include most core

metadata, such as component names, unique identifiers, and dependency relationships, but none fully meet NTIA minimum requirements due to missing supplier information. Version information is generally available for top-level dependencies, while transitive dependencies are only listed when their parent dependencies include versions. License information is inconsistently reported across dependency components.

RQ2: How consistent are GitHub SBOMs compared to SBOMs from other generators? This question explores how GitHub SBOMs differ from those produced by three widely used tools in terms of metadata coverage and consistency across ecosystems. We found that GitHub SBOMs generally report more dependencies and provide version and license information more frequently than those generated by Trivy and Syft. However, SBOMs generated by the Microsoft SBOM Tool tend to more consistently include version information.

RQ3: How useful are GitHub SBOMs compared to SBOMs from other generators in vulnerability tracking? This question evaluates how GitHub SBOMs compare against SBOMs generated by other tools in supporting automated vulnerability tracking. We found that GitHub and Microsoft SBOMs tend to list more components, and more frequently include unique identifiers (PURLs) that can be used for vulnerability tracking. SBOMs generated by Syft and Trivy, on the other hand, frequently include components that lack precise version information. Consequently, GitHub SBOMs can be mapped to more vulnerabilities than other tools' SBOMs across nearly all languages. However, the accuracy of vulnerability detection depends on whether exact version information is provided for the components.

1.2 Research Statement

Motivated by GitHub's position as the largest host of open-source software and the ease with which developers can generate SBOMs using its native functionality, this thesis investigates whether GitHub SBOMs provide the metadata required for security and compliance tasks. The research statement is as follows:

Automatically generated SBOMs are valuable if they provide complete, consistent, and actionable metadata. GitHub generates SBOMs using a common approach across software ecosystems that differ substantially in how dependencies are declared and managed. This thesis systematically evaluates the quality of GitHub SBOMs at scale by examining whether they satisfy established minimum metadata requirements, comparing them with other widely used SBOM generation tools, and assessing their reliability for automated vulnerability tracking. It also investigates the ecosystem-level factors that contribute to any deficiencies identified. By providing a large-scale empirical evaluation of GitHub SBOMs, this thesis advances our understanding of whether these SBOMs can be relied upon for software supply chain security and identifies the limitations that affect their practical use.

1.3 Thesis Overview

In this section, we provide an overview of the work presented in this thesis and highlight the main themes of each chapter.

Chapter 2: Background and Related Work. This chapter introduces the key concepts underlying this thesis. We describe what an SBOM is and the benefits it provides, the SPDX format used by GitHub-generated SBOMs, and the terminology we use to distinguish top-level from transitive dependency components. We then describe how GitHub generates SBOMs from a repository's dependency graph, and detail the NTIA minimum elements that an SBOM should provide to be considered compliant. The chapter closes by reviewing prior empirical studies on SBOM adoption and tooling, work specifically examining compliance with NTIA requirements, studies on the broader challenges of SBOM implementation, and research on the role of SBOMs in software security and supply-chain transparency, positioning our study relative to the two most closely related empirical evaluations of SBOM generation tools.

Chapter 3: Methodology. This chapter details our study design. We describe how we selected 10,000 repositories across ten programming languages, how we generated SBOMs using the GitHub SBOM Tool, Syft, Trivy, and the Microsoft SBOM Tool, and the environment and benchmarking

approach used throughout our analysis.

Chapter 4: Results. This chapter presents our findings for all three research questions. We first assess the completeness of GitHub-generated SBOMs with respect to NTIA compliance, version coverage, and license availability (RQ1). We then compare GitHub SBOMs against the three other SBOM generation tools in terms of component detection and metadata completeness (RQ2). Finally, we evaluate the practical usefulness of the four tools for automated vulnerability tracking using OSV.dev (RQ3).

Chapter 5: Discussion. This chapter discusses four aspects of our results in greater depth: an accuracy evaluation of the four tools against a manually constructed ground truth, the component-level agreement between the GitHub SBOM Tool and the Microsoft SBOM Tool, the potential causes of missing version information in Python projects, and the reasons why license information is seldom reported across SBOM tools and package managers.

Chapter 6: Threats to Validity. This chapter discusses the threats to the validity of our study, including the generalizability of our results, internal threats related to tool execution, reproducibility limitations across the four SBOM tools, and threats to the validity of our per-language conclusions.

Chapter 7: Conclusion. Finally, this chapter summarizes the thesis's key findings and outlines directions for future research.

1.4 Thesis Contributions

The main contributions of this thesis are as follows:

- We conduct the first comprehensive empirical study of GitHub-generated SBOMs using 10,000 repositories (1,000 repositories per programming language) across 10 programming languages, evaluating SBOM quality and completeness against the NTIA minimum element requirements.
- We compare four major SBOM generators, revealing variations in version coverage, availability of unique identifiers, and license reporting across programming ecosystems.
- We evaluate the practical usefulness of SBOMs for vulnerability tracking, since one of the

primary purposes of an SBOM is to give software consumers visibility into a software's components so they can understand whether the software they use may expose them to vulnerabilities through those components.

- We publish our replication package[1] to help foment more research on the tooling and SBOM quality topic.

1.5 Related Publications

The work presented in this thesis has been submitted to the following venue for review:

- Kawsar Ahmed Bhuiyan, Mohamed Bilel Besbes, Rachna Raj, Adam Al Assil, and Diego Elias Costa. Beyond Compliance: A Large Scale Study on the Completeness and Consistency of the GitHub SBOMs. *Submitted to Empirical Software Engineering*.

[1] `https://doi.org/10.5281/zenodo.18883005`

Chapter 2

Background and Related Work

2.1 What is an SBOM

A Software Bill of Materials (SBOM) is a formal record that contains details and supply chain relationships of the various components used in building software [17]. An SBOM lists all software components, including relevant component metadata, and describes the relationships between them. For example, an SBOM for a web application might list open-source libraries such as React, Lodash, and Axios, including their versions, licenses, and their relationships within the application's dependency hierarchy. SBOMs offer several key benefits that enhance software development and security processes [16]. They provide clear visibility into all software components and their dependencies, enabling teams to manage and track both direct and transitive dependencies efficiently. This transparency helps organizations quickly identify and prioritize vulnerabilities, improving incident response and risk management. Additionally, SBOMs support license compliance by allowing legal teams to verify that all components adhere to organizational policies. Beyond internal benefits, SBOMs also create a competitive advantage for vendors, as they are increasingly required in procurement processes and signal a commitment to software quality and security [18].

The SPDX format. SPDX (System Packet Data Exchange, formerly Software Packet Data Exchange) is an open standard for communicating Software Bill of Materials information, including provenance, licensing, security, and related metadata [19]. By providing a common format, SPDX

helps reduce redundant efforts across organizations and communities, facilitating the sharing of critical data and thereby streamlining compliance, security, and reliability processes [20]. Figure 2.1 visually outlines these key attributes, categorizing them into mandatory and optional components in the SPDX v2.3 format. The core required attribute in any SPDX document is the Creation Information section, which provides the foundational metadata necessary for the document's identification and compatibility with various tools. Additionally, the SPDX format supports several optional sections that allow for a detailed description of the software and its components, including Package Information, File Information, Snippet Information, Other Licensing Information, Relationships, Annotations, and Review Information.

Terminology: Components, packages, and dependencies. SPDX defines in the *package information* of an SBOM all components that compose the software system. In this thesis, we use the term *component* to refer to all components in a software project. However, when an analysis discriminates between the types of dependency components, we refer to them either as *top-level dependencies* or *transitive dependencies* of a software project.

2.2 GitHub & SBOM Generation

GitHub holds a pivotal role in contemporary software development due to its widespread adoption as a collaborative platform for version control, code sharing, and project management. It serves as a central repository for millions of open source repositories, facilitating efficient teamwork and transparency. While numerous tools exist for generating Software Bill of Materials (SBOMs), the GitHub SBOM Tool stands out for its native integration within the GitHub ecosystem. This integration relies on repositories' dependency graphs, which summarize manifest and lock files, along with dependencies submitted via the dependency submission API. A dependency graph provides information on dependencies' versions, licenses, manifest sources, and known vulnerabilities, including transitive dependency paths when supported. Importantly, the SBOM generated from this data stays synchronized with the repository's current dependencies and can be exported in the standard SPDX format via the GitHub UI or REST API [14]. This reduces manual work and helps maintain an up-to-date view of a project's software supply chain.

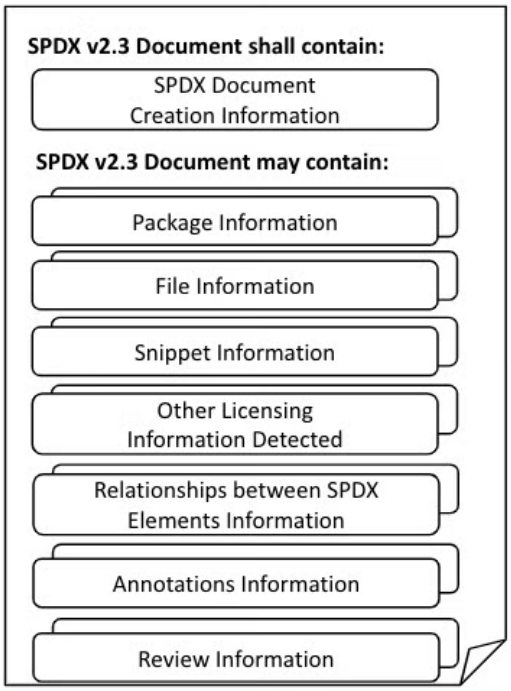

Figure 2.1: SPDX Document Overview [21]

```
{
  "sbom": {
    "SPDXID": "SPDXRef-DOCUMENT",
    "spdxVersion": "SPDX-2.3",
    "creationInfo": {
      "created": "2021-09-01T00:00:00Z",
      "creators": [
        "Tool: GitHub.com-Dependency-Graph"
      ]
    },
    "name": "github/example",
    "dataLicense": "CC0-1.0",
    "documentNamespace": "https://spdx.org/...",
    "packages": [
      {
        "name": "rails",
        "SPDXID": "SPDXRef-Package",
        "versionInfo": "1.0.0",
        "downloadLocation": "NOASSERTION",
        "filesAnalyzed": false,
        "licenseConcluded": "MIT",
        "licenseDeclared": "MIT",
        "copyrightText": "Copyright (c) 1985",
        "externalRefs": [
          {
            "referenceCategory": "PACKAGE-MANAGER",
            "referenceType": "purl",
            "referenceLocator": "pkg:gem/rails@1.0.0"
          }
        ]
      }
    ],
    "relationships": [
      {
        "relationshipType": "DEPENDS_ON",
        "spdxElementId": "SPDXRef-Repository",
        "relatedSpdxElement": "SPDXRef-Package"
      }
    ]
  }
}
```

Listing 2.1: GitHub-generated SBOM Example [22]

Listing 2.1 illustrates a *GitHub SBOM* (short for GitHub-generated SBOM) in the SPDX v2.3 JSON format. It shows how the GitHub SBOM Tool captures SPDX metadata such as the SPDX version, creation information, and document namespace. The packages section details a dependency (rails 1.0.0) along with its license (MIT), and package URL. The relationships section explicitly records dependency relationships, indicating which packages the repository depends on.

2.3 NTIA compliance

The National Telecommunications and Information Administration (NTIA) has established guidelines for Software Bill of Materials (SBOM) compliance [13,23,24]. As detailed in Table 2.1, NTIA compliance requires SBOMs to include seven fundamental data fields that collectively enable comprehensive software component tracking and risk assessment. These minimum elements serve distinct but interconnected purposes in establishing supply chain transparency. The significance of each field, as outlined in the table, demonstrates how NTIA's requirements address critical aspects of software security management, from tracing distribution sources and maintaining component inventories to identifying vulnerable versions and understanding inter-component dependencies. Additionally, compliant SBOMs must support automation and be machine-readable, typically delivered in standardized formats such as SPDX, CycloneDX, or SWID tags [25]. This regulatory framework represents a significant shift toward mandating supply chain transparency, particularly for software vendors serving federal government clients, while establishing a foundation for broader industry adoption of systematic software component tracking and risk assessment practices that leverage the structured data fields specified in NTIA's minimum requirements.

2.4 Related Work

In this section, we discuss the related work, grouped into four themes. Section 2.4.1 reviews empirical studies on SBOM adoption and tooling. Section 2.4.2 covers work specifically examining compliance with NTIA requirements. Section 2.4.3 discusses studies on the broader challenges of SBOM implementation. Finally, Section 2.4.4 reviews research on the role of SBOMs in software security and supply-chain transparency.

Table 2.1: NTIA Minimum Requirements for SBOMs [13]

Data Field	**Description**	**Reason of significance**
Supplier Name	The name of an entity that creates, defines, and identifies components.	It helps trace the distribution source for a specific package as the same component may be supplied through different suppliers (e.g., maven repository, Gradle central repo).
Component Name	Designation assigned to a unit of software defined by the original supplier.	It is crucial to maintain an inventory list of all components used in a software such as packages and files. In GitHub's generated SBOMs, mainly packages are used.
Version of the Component	Identifier used by the supplier to specify a change in software from a previously identified version.	Versions ensure compatibility and up-to-dateness as it identifies any changes to the software. Therefore, having it is important because we can identify which versions are vulnerable and need updates.
Other Unique Identifiers	Other identifiers that are used to identify a component, or serve as a look-up key for relevant databases.	Identifying components could be confusing so having a unique identifier for each component would make things easier.
Dependency Relationship	Characterizing the relationship that an upstream component X is included in software Y.	Dependency relationships specify different relationship types between components of the software such as which packages depend on others.
Author of SBOM Data	The name of the entity that creates the SBOM data for this component.	Identifies who created the SBOM as there are different tools such as GitHub's dependency graph.
Timestamp	Record of the date and time of the SBOM data assembly.	Timestamps verify a record of when the SBOM was created for a specific software. It is important because as projects get updated, the SBOM should also change.

2.4.1 Empirical Studies on SBOM

Empirical research on SBOM adoption has provided useful insights into the practices and challenges observed across industry and open-source ecosystems. Xia et al. conducted an empirical study based on interviews with practitioners and a survey of professionals to examine current SBOM practices, available tool support, and major concerns surrounding SBOM usage [26]. Their findings indicate limited adoption in practice. Even among projects that generate SBOMs, there is no clear consensus on what information to include, despite official recommendations and guidelines, underscoring the need for clearer expectations and stronger standardization.

These observations are consistent with evidence reported in the Linux Foundation's SBOM readiness survey, which analyzed responses from 412 organizations worldwide [27]. The survey revealed notable gaps in organizational familiarity with SBOMs, as well as uncertainty regarding SBOM production and consumption strategies. In addition, many respondents expressed skepticism about whether SBOM requirements are being broadly adopted across the software industry, raising concerns about alignment with emerging regulatory and policy initiatives.

Beyond adoption awareness, Xia et al. further identified significant limitations in existing SBOM tooling, particularly for SBOM consumption and vulnerability handling, underscoring the need for more robust, interoperable, and enterprise-ready solutions [26]. The authors also emphasized the importance of validation and verification mechanisms to ensure the reliability and trustworthiness of SBOM data, especially given the risk of incomplete or manipulated metadata. Together, these findings highlight the need for continued improvements in SBOM tooling and practices to support meaningful adoption.

Nocera et al. focused specifically on open-source projects hosted on GitHub, uncovering a growing trend in SBOM adoption [15]. Despite this increase, the overall uptake remains limited, particularly in smaller projects. Their comparison of SPDX and CycloneDX formats highlighted that while SPDX is widely adopted, CycloneDX offers superior support for security-related features, which is a critical consideration for projects prioritizing security.

One of the closest related works is the work of Yu et al., who compared the quality of SBOM generation across GitHub Dependency Graph, Syft, Trivy, and Microsoft SBOM Tool [28]. The

study covered 7,876 open-source repositories spanning nine programming languages. The authors compared the tools' outputs by examining the number of detected packages, measuring pairwise overlap using Jaccard similarity over (name, version) dependency sets, and quantifying the presence of duplicate packages within individual SBOMs. Although this work is similar to ours, their analysis focused primarily on how different tools vary in reporting the number of dependencies in SBOMs, whereas we focused on how the tools differ in reporting key metadata, such as component version, licensing, and compliance with NTIA requirements.

Another close work to ours is Wang et al. [29]. They present an empirical study of 3,287 repositories across four languages (C, C++, Java, and Python), focusing on three dimensions: structural compliance with SBOM standards, inter-tool consistency, and information accuracy. Consistent with our results, they found inadequate compliance with policy requirements, poor inter-tool consistency, and low license reporting rates. Our study is complementary in multiple ways. We cover more programming languages, which enables us to draw parallels across other ecosystems. We also include an in-depth investigation into the potential root causes of missing license information across ecosystems and evaluate the practical utility of SBOMs for automated vulnerability tracking. The work of Wang et al. [29] was published recently, and was conducted concurrently with our study. We believe that both studies offer complementary perspectives on the state of SBOM tooling.

Table 2.2 summarises the key dimensions along which our study differs from the two most directly related works, [28] and [29]. The table highlights that while all three studies share a concern for SBOM quality and inter-tool comparison, each addresses a distinct set of research questions, and our work uniquely combines the broadest language coverage, a focus on the GitHub-native SBOM Tool, end-to-end vulnerability tracking evaluation, and a systematic root-cause analysis of missing metadata at the package-manager level.

2.4.2 Compliance with NTIA Requirements

Compliance with NTIA requirements is a pivotal aspect of SBOM implementation, and several studies have explored the associated challenges and benefits. Zahan et al. identified the top five benefits and challenges of aligning SBOMs with NTIA guidelines [18]. Their findings suggest that

Table 2.2: Comparison of our study with the two most closely related empirical SBOM studies. ✗ denotes not addressed.

Dimension	**Yu et al. [28]** *DSN 2024*	**Wang et al. [29]** *ACM TOSEM 2026*	**This Work** *Submitted to EMSE 2026*
Study scale	7,876 repos; 9 languages; 4 tools	3,287 repos; 3 languages; 6 tools	**10,000 repos; 10 languages; 4 tools**
SBOM compliance assessment	✗	extended NTIA field set	all 7 NTIA minimum elements, per language
Metadata completeness analysis	version only; pinned vs. unpinned	version, license, PURL; field-level per tool	version, license, PURL; top-level & transitive, per tool & language
Inter-tool comparison	component counts + Jaccard similarity	triple-factor matching; field-level consistency	counts, version, license; Kruskal-Wallis + Dunn per language
Ground-truth accuracy evaluation	Python only	Python only; 100 repos	Python, Java, JS; 200 repos each
Vulnerability tracking utility	✗	✗	PURL-based via OSV.dev; effect of version precision on detection, per tool & language
Root-cause analysis of missing metadata	parser limitations & metadata file constraints	standard ambiguity, scope definition	7 causes at package manager level

while SBOMs significantly enhance transparency and security, the complexity of achieving compliance poses a substantial hurdle. Similarly, Torres et al. conducted a study through which multiple SBOM generating tools were assessed according to NTIA requirements to see how compliant they are with these latter [30]. Only 1% of SBOMs coming from the assessed Docker images are fully NTIA compliant.

Nocera et al. analyzed 119 SBOMs—89 using CycloneDX and 30 using SPDX—spanning 84 projects and 22 owners [31]. In their assessment, the authors noted that the analyzed SPDX SBOMs contained all NTIA minimum data fields *at least once*, with the supplier name and dependency relationship appearing in only a single SBOM. Based on this, they concluded that 3% $(1/30 \times 100)$ of the SPDX SBOMs satisfied all NTIA minimum data fields.

2.4.3 Challenges in SBOM Implementation

Research into the challenges of SBOM implementation has identified a range of technical and organizational obstacles. Stalnaker et al. detailed 12 major challenges in creating and using SBOMs, including tool deficiencies, domain-specific challenges, and difficulties in integrating SBOMs into

existing workflows [32]. These findings are echoed by Torres-Arias et al., who argue that the development of quality measurement mechanisms for SBOMs is crucial for enhancing their effectiveness [30]. The study emphasizes that without robust mechanisms to assess SBOM quality, their potential benefits in security and transparency may not be fully realized. Dalia et al. published a paper through which they conducted an analysis of the challenges holding back SBOM adoption [33]. It further conducted a comparative analysis between multiple SBOM generation tools based on the features considered, based on the challenges initially stated in the paper. Bi et al. proposed a study through which the first comprehensive classification of SBOM-relevant issues and potential solutions has been provided, along with an identification of the different phases of the SBOM lifecycle and their characteristics [34]. In this classification, issues were correlated with various stages of the SBOM lifecycle, offering a structured analysis that serves as a guide for developers. This analysis aids in understanding the challenges associated with SBOMs and offers recommendations for effectively integrating SBOMs to address development issues in real-world scenarios. Additionally, the study identifies gaps in the existing production and usage of SBOMs, highlighting shortcomings in current practices. Based on these insights, the authors suggest future research directions aimed at improving SBOM quality and adoption, providing a roadmap for advancing the field.

2.4.4 Impact of SBOMs on Software Security and Transparency

SBOMs are increasingly recognized for their role in improving software supply chain security and transparency. Xia et al. explored the integration of blockchain technology into SBOMs as a means of enhancing security by providing a tamper-resistant mechanism for SBOM sharing [9]. This study suggests that blockchain can significantly improve the trustworthiness of SBOMs, particularly in environments where security is paramount. Similarly, Carmody et al. examined the potential of SBOMs in the medical technology supply chain, highlighting their ability to improve transparency and trust among stakeholders [35]. Sharma et al. introduced SBOM.exe, a tool designed to detect the malicious usage of dynamic features in Java by verifying the binary integrity of an application's dependencies at runtime [36].

Chapter 3

Methodology

Our goal is to study the completeness and consistency of SBOMs generated by GitHub across 10 programming ecosystems. Figure 3.1 illustrates our methodology. We begin by selecting 1,000 projects from each programming language, aiming to capture active, highly popular, and collaborative software repositories (Section 3.1). For each repository, we generate a GitHub SBOM, yielding a total of 10,000 SBOMs (Section 3.3). To enable comparative analysis, we also generate SBOMs for the same repositories using three widely used tools: Syft, Trivy, and the Microsoft SBOM Tool (Section 3.3). We evaluate GitHub SBOMs for NTIA minimum compliance, version coverage, and license availability (RQ1, Section 4.1). We then compare GitHub SBOMs with SBOMs generated by other tools to assess differences in metadata completeness and dependency detection (RQ2, Section 4.2). Finally, we examine the practical usefulness of SBOMs for detecting known vulnerabilities (RQ3, Section 4.3). Together, these analyses provide a comprehensive view of SBOM completeness, metadata quality, and practical utility across programming languages and tooling.

3.1 Project Selection

To analyze SBOMs generated by GitHub, we target *open-source* repositories that are *widely used* and *actively maintained* by multiple contributors. These projects are more likely to reflect mature development practices and realistic software supply-chain behaviours. This choice is motivated by prior evidence that GitHub contains a substantial amount of repositories that are personal

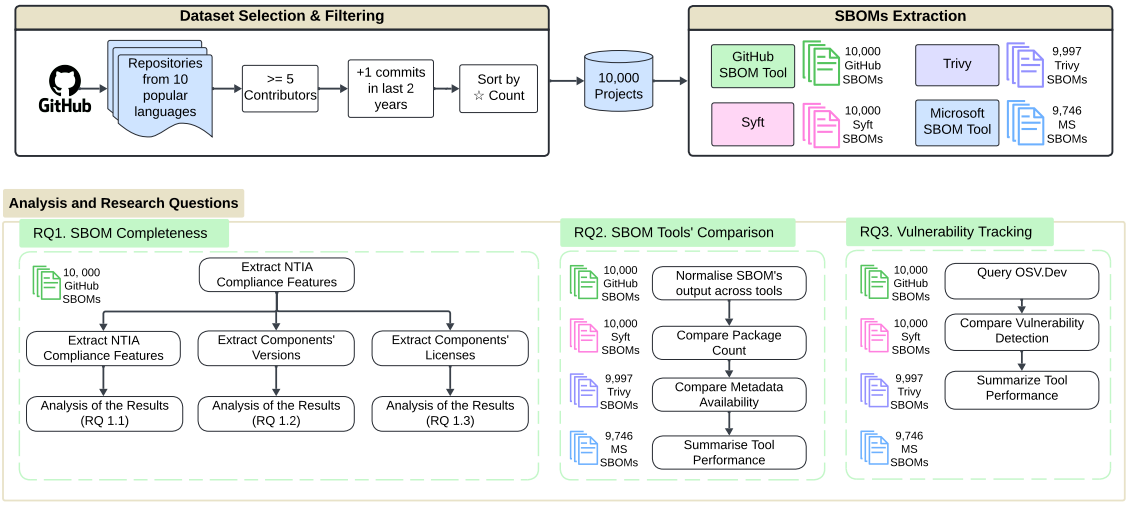

Figure 3.1: Methodology overview of this study.

or inactive, and that naive sampling can therefore lead to datasets that are not representative of collaborative software development [37].

Within this population, we strategically select projects from ten widely used programming languages: C, C++, C#, Java, PHP, Python, JavaScript, Go, Rust, and Swift. These languages were chosen because they are consistently reported among the most popular and widely adopted languages across complementary popularity indicators, including the TIOBE Programming Community Index [38], GitHub's Octoverse language rankings [39], and large-scale developer surveys from Stack Overflow [40].

Step 1: Contributor Threshold. We select only repositories with at least **five unique contributors**. This criterion helps us focus on *collaborative* and *community-driven* projects (instead of personal/toy repositories), which are more likely to reflect real-world development and maintenance practices. Similar contributor-based thresholds are commonly used in prior empirical studies in software engineering [41, 42].

Step 2: Recent Maintenance. The software repository must have shown at least one commit within the two years prior to our data extraction. This condition ensures that selected projects are minimally maintained. For the filtering in Steps 1 and 2, we use the SEART GitHub Search (GHS) platform [43], a web-based tool designed to support reproducible sampling of GitHub repositories

for mining software repositories studies. SEART-GHS maintains a continuously updated dataset of GitHub repository metadata and provides a query interface that enables filtering projects based on multiple criteria, such as programming language, repository activity, popularity indicators, and licensing information [44]. SEART-GHS has been adopted in prior empirical software engineering studies to select representative sets of actively developed repositories [45–47], demonstrating its suitability for constructing reliable and reproducible datasets.

Step 3: Popularity by Stars. We select the **top 1,000 most popular projects per programming language** based on their GitHub star count, resulting in a total of **10,000 projects across 10 programming languages**. GitHub allows users to *star* repositories to express interest in or appreciation of a project. Prior empirical research has shown that the number of stars is commonly used as a proxy for project popularity, where repositories with higher star counts tend to receive greater community attention and adoption [48]. Following this established practice, we prioritize projects with higher star counts to capture influential and widely adopted software systems. This focus on popular repositories increases the likelihood that the analyzed projects reflect mature development practices and have broader ecosystem impact, thereby improving the generalizability and relevance of our findings. Following these steps, we selected 1,000 repositories for each of the 10 programming languages, resulting in a total dataset of 10,000 repositories. We present the distribution of the selected projects across different characteristics in Figure 3.2.

3.2 Selection of SBOM Generation Tools

To compare the quality of GitHub SBOMs, we select the other widely used SBOM generation tools:

- **Syft:** An open-source CLI tool developed by Anchore that generates SBOMs from container images, filesystems, and source directories, supporting multiple formats including SPDX and CycloneDX [49].
- **Trivy:** A security and compliance scanner maintained by Aqua Security that can produce SBOMs as well as detect vulnerabilities in container images, filesystems, and Git repositories [50].

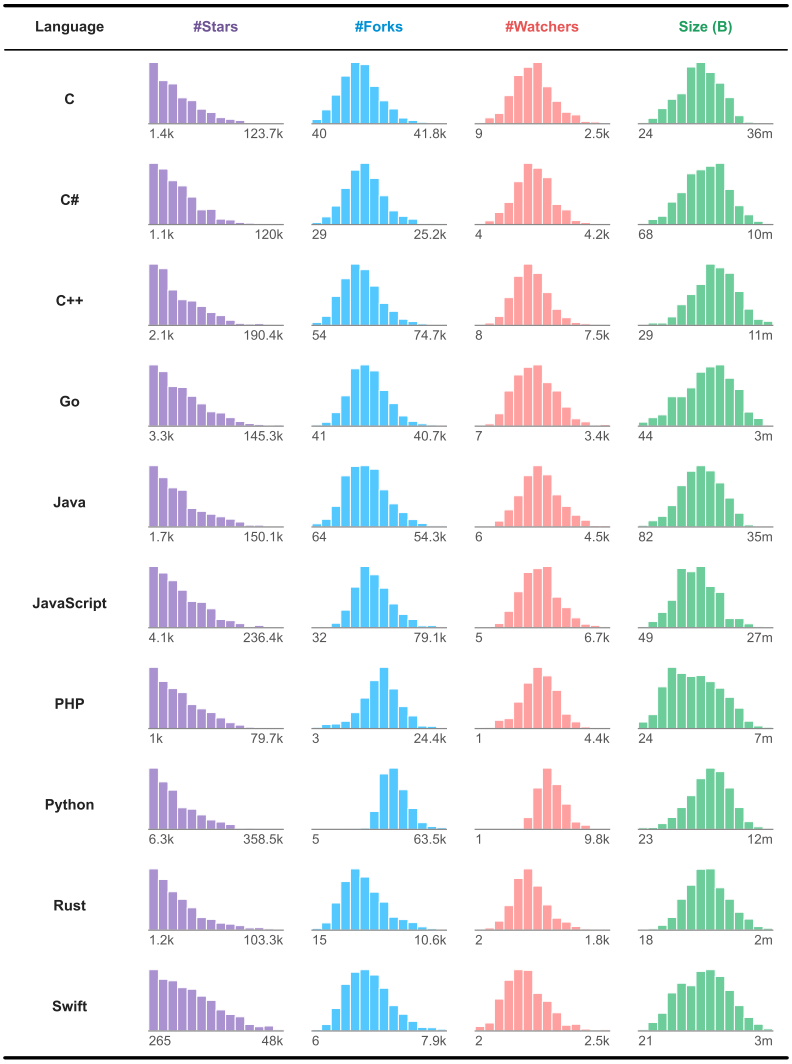

Figure 3.2: Statistical Distribution of the Selected 10,000 Repositories

- **Microsoft SBOM Tool:** A free command-line tool provided by Microsoft that generates SBOMs for a wide range of project types, supporting multiple programming languages and producing output in SPDX and CycloneDX formats [51].

We selected these tools because they are free, open-source, and support multiple operating systems (Linux, Windows, and macOS). These tools are also frequently used in previous studies [28, 30] and industry overviews [52, 53], demonstrating their relevance and adoption.

3.3 SBOM Generation

Upon selecting our dataset of the most popular 10,000 open-source repositories across 10 programming languages, we aim to export their SBOMs using the GitHub SBOM generator, Trivy, Syft, and Microsoft SBOM Tool.

GitHub SBOM Tool. To extract SBOMs generated from the recently added GitHub SBOM Tool, we use the GitHub REST API for SBOMs [22] to programmatically generate the SBOM for each selected repository by issuing a `GET` request to the `/repos/{owner}/{repo}/dependency-graph/sbom` endpoint, which returns an SPDX-compatible SBOM derived from GitHub's dependency graph when enabled. All of the SBOMs were obtained in SPDX JSON format, as GitHub's SBOM REST API currently exports SBOMs in this format [22]. For each language, we initiated SBOM extraction starting from the most popular repositories and proceeded down the list. Since GitHub requires the dependency graph to be enabled in a repository to export an SBOM [14], we received a 404 error when attempting to generate SBOMs via the REST API for repositories where the dependency graph was disabled by the owners. In such cases, we skipped the repository and continued to the next most popular one. This iterative process was repeated until we successfully obtained 1,000 valid SBOMs per language.

Extraction from Local Repositories. All three SBOM generator tools, Trivy, Syft, and Microsoft SBOM Tool, provide support for SBOM generation using local repositories [54–56]. Following the GitHub-based SBOM generation, all 10,000 repositories were cloned to local storage immediately thereafter, and all local SBOM generation runs were performed against these same cloned snapshots. By cloning directly after the GitHub phase, the local snapshots reflect the same repository state that was present during GitHub SBOM generation, minimizing the risk of source divergence between the two extraction methods. Each tool was then executed against the same locally cloned repository snapshot, ensuring identical source code, dependency manifests, and project structures were analyzed across all three tools. Each tool has parameters that control the sensitivity of the SBOM generation. We configured each tool to run under the most permissive license-discovery settings available to obtain the maximum possible license coverage

Trivy. For Trivy, SBOMs were generated from the locally cloned repositories using Trivy's filesystem scanning mode. For each repository, we executed Trivy with SPDX JSON output enabled. We enabled `--license-full`, which instructs it to scan not only component metadata but also the full contents of each file for embedded license expressions. Trivy successfully generated SBOMs for 9,997 repositories. Three failures occurred due to excessive analysis time on large multi-module Java projects and a stack-overflow issue triggered by its XML-based Go module parsing.

Syft. Syft operates exclusively on local inputs such as source code directories, container images, or packaged artifacts [55]. For each cloned repository, Syft was executed with the repository directory as input and configured to emit SPDX JSON output. Syft successfully generated SBOMs for all 10,000 repositories in the dataset.

Microsoft SBOM Tool. The Microsoft SBOM Tool similarly requires local repository inputs [56]. For each repository, SBOM generation involved creating a temporary manifest directory and executing the tool with repository metadata parameters. The Microsoft SBOM Tool internally queries the ClearlyDefined [57] API, an open-source crowdsourced database of license and attribution metadata, to retrieve license information. The resulting SPDX JSON files were extracted from the generated manifest directory. The Microsoft SBOM Tool exhibited a higher failure rate, particularly on repositories containing recursive symbolic-link structures or missing build-generated files. These failures stem from limitations in its hashing mechanism, which cannot correctly handle symlink loops or absent files, resulting in a total of 9,746 successfully generated SBOMs.

This chapter described the study design shared across all three research questions. The specific analytical approach for each research question is presented alongside its results in Chapter 4.

Chapter 4

Results

In this chapter, we present the results of our study. For each research question, we describe our approach and report the corresponding findings.

4.1 RQ1: How compliant and complete are GitHub SBOMs?

Understanding the completeness of GitHub-generated SBOMs is essential because the usefulness of any SBOM depends on the accuracy and availability of the information it provides [25]. While specific needs may vary by use case, all applications require a consistent and systematic approach to defining and identifying software components and their relationships. An SBOM typically includes both the primary software component and its dependency components. In this study, we focus on the latter.

We assess the completeness of the generated SBOMs with respect to three main aspects: (1) the NTIA minimum requirements (see Table 2.1), (2) the presence of version information for dependencies, and (3) the availability of licensing information for dependencies. Accordingly, we structure **RQ1** into three sub-questions:

- **RQ 1.1:** Do GitHub SBOMs satisfy the NTIA minimum requirements?
- **RQ 1.2:** To what extent do GitHub SBOMs include **version information** for dependencies?
- **RQ 1.3:** To what extent do GitHub SBOMs provide **licensing information** for dependencies?

4.1.1 Approach

To evaluate the completeness of GitHub SBOMs, we analyze each SBOM and aggregate the results by programming language to identify ecosystem-level patterns.

Assessing NTIA Compliance of GitHub-Generated SBOMs. According to the NTIA guidance on the Minimum Elements for an SBOM, an SBOM should list all primary components along with their dependencies [13]. We analyze each SBOM file to identify the NTIA-required elements (listed in Table 2.1) using the corresponding SPDX field names and specification references.

- **Component Name:** The primary component, i.e., the software system for which the SBOM was generated, is identified from the top-level `name` field of the SBOM document. Dependency components' names are found by inspecting the `name` field within each entry of the `packages` array.
- **Version of the Component:** The version of each component—including the primary component and third-party library packages—is extracted from the `versionInfo` field located in its respective entry within the `packages` array.
- **Supplier Name:** The supplier is extracted by checking the `PackageSupplier` or `supplier` field within each package entry.
- **Other Unique Identifiers:** Unique identifiers for components are found in the SPDXID field, while the SBOM document itself is uniquely referenced through the `documentNamespace` field. Examples of commonly used unique identifiers are Common Platform Enumeration (CPE), Software Identification (SWID) tags, and Package Uniform Resource Locators (PURL) [13]. We tried to look for CPE, SWID or PURL by inspecting the `externalRefs` field of each entry of the `packages` array. We examine the `externalRefs` field of each entry in the `packages` array and consider the presence of a unique identifier if at least one of CPE, SWID, or PURL is found.
- **Dependency Relationship:** When present, dependency relationships between components are described in the `relationships` array.

- **Author of SBOM Data:** The authoring entity is identified through the `creators` field under the `creationInfo` section. This may include the names of tools, individuals, or organizations responsible for generating the SBOM.
- **Timestamp:** The timestamp indicating when the SBOM was created is retrieved from the `created` field within the `creationInfo` section.

NTIA recommends that SBOMs include all top-level dependencies with sufficient detail to enable the recursive discovery of transitive dependencies [13]. Therefore, in this analysis, we focus primarily on the top-level dependencies for each project and verify whether each of them includes required fields such as `name`, `supplier`, `versionInfo`, and a unique identifier. We determine the top-level dependencies by examining the `relationships` array and selecting entries where the `spdxElementId` matches this primary identifier.

Comparison of Version Coverage of Top-Level and Transitive Dependencies. Beyond just the NTIA compliance, which considers just the top-level dependencies, it is also beneficial to include version information for transitive dependencies. Transitive dependencies are known to introduce security vulnerabilities, licensing risks, or compatibility issues [58,59]. To identify transitive dependencies, we trace the dependencies from each top-level dependency to the components it depends on. After identifying both primary and transitive dependencies, we inspect the `versionInfo` field in their corresponding entries within the `packages` array to verify whether valid version information is provided.

Identifying Components' License Availability. For each SBOM, we checked whether license information was available for the top-level dependencies and transitive dependencies. Within the `packages` array, the `licenseConcluded` and `licenseDeclared` fields were checked for all the dependencies. We measured the share of components with available license information among top-level dependencies, transitive dependencies, and all dependencies combined (the union of both). These dependency-level percentages were then averaged by language to determine overall license availability trends.

4.1.2 Do GitHub-generated SBOMs satisfy the NTIA minimum requirements?

We present the NTIA compliance results for all analyzed SBOMs, summarized in Table 4.1, including a breakdown by element, attribute, and programming language. We observe the following:

Finding 1. No GitHub SBOM is fully compliant with the NTIA minimum requirements. To be considered fully compliant, an SBOM must satisfy all defined minimum elements, and in our analysis, that was obtained by no SBOM generated by GitHub. The most critical gap is the absence of supplier information—0% of the SBOMs include supplier data for either the primary component or any top-level dependencies. Supplier information is an important information as it specifies the name of the entity that creates, defines and identifies components. For example, *Guava* is a Java library developed and distributed by *Google*; an SBOM that lists *Guava* as a dependency should therefore identify *Google* as its supplier. This universal omission of supplier data leads to 0% full compliance with the NTIA minimum requirements across all evaluated languages.

Finding 2. GitHub SBOMs show strong compliance in core metadata fields. Despite the absence of supplier information, all other NTIA-required fields are widely represented and compliant across SBOMs. Every SBOM includes the name and version of the primary component, as well as their unique identifier. In addition, 100% of SBOMs provide dependency relationships for all listed components, author metadata, and timestamps. Additionally, all SBOMs include the name and unique identifiers for all the top-level dependencies. For GitHub SBOMs, all the top-level dependencies include SPDXID as well as PURL as their unique identifier.

Finding 3. Dependency version coverage in projects varies significantly per programming language. The level of compliance varies substantially across projects written in different programming languages. Swift exhibits the highest coverage, with 98.8% of SBOMs providing version information for all top-level dependencies, followed closely by JavaScript (96.2%), Go (93.5%), and Rust (91.8%). In contrast, Java and Python show the lowest compliance in this area, with respectively only 41.4% and 29.9% of their SBOMs including version information for all the top-level dependencies. C, C#, C++, and PHP fall in the mid-to-high range, with coverage between 75.0% and 83.3%.

Table 4.1: Compliance of SBOMs Across Different Languages with NTIA's Minimum Requirements

Element	Attribute	C	C#	C++	Go	Java	JS	PHP	Python	Rust	Swift
Primary Software Component	Supplier	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
	Name	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
	Version	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
	Unique ID	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
All Top-Level Dependency Components	Supplier	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
	Name	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
	Version	83.3%	79.5%	75.0%	93.5%	41.4%	96.2%	82.9%	29.9%	91.8%	98.8%
	Unique ID	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
All Components	Dependency Relationships	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
SBOM	Author	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
	Timestamp	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
Overall Compliance		0%	0%	0%	0%	0%	0%	0%	0%	0%	0%

4.1.3 Do GitHub-generated SBOMs include version information for dependencies?

We present the results of our analysis in Figure 4.1. We observe significant variation in the consistency with which generated SBOMs contain dependency version information across languages. It is important to note that the values in the table differ significantly from those in our analysis in RQ1. RQ1 focuses on NTIA compliance and shows the share of projects that include version information for all top-level dependencies, while in this analysis, we are interested in the average distribution of version information across projects.

Finding 1. With the exception of Python and Java projects, GitHub-generated SBOMs, on average, include more than 94% of versioned dependencies. We show in Figure 4.1 that the presence of versioned dependencies encompasses at least 94% of a project's dependencies across most of the evaluated programming languages. The share of versioned dependencies exceeds 99.5% in projects in Go, Rust, and Swift, showing the strongest support across the ten languages.

The exception to this rule is projects in Java and Python. Java projects exhibit an average of 83% of versioned dependencies, while Python projects show only 78% of versioned dependencies. These results are in line with our results in RQ1 (Table 4.1), where we showed that Java and Python

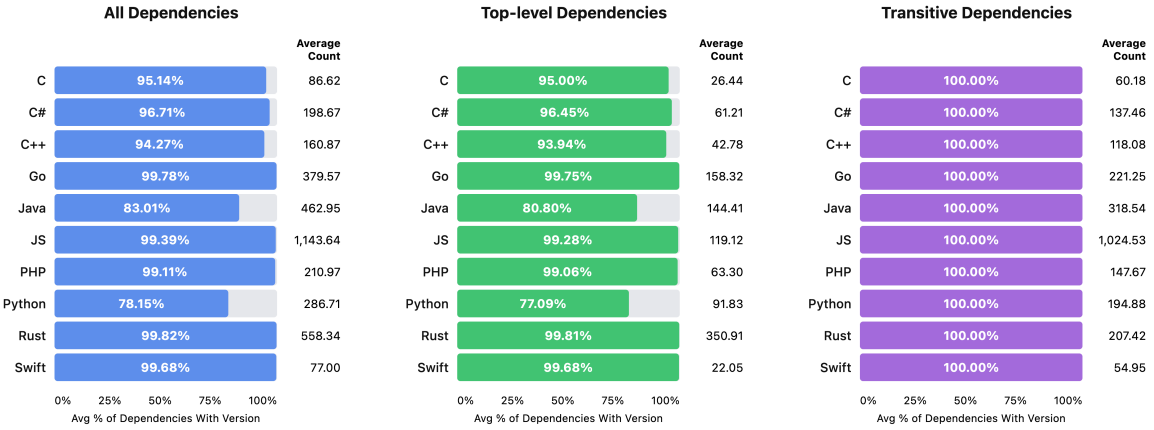

Figure 4.1: Average version information of dependencies per project.

projects exhibited the least share of projects with version included in all top-level dependencies. C, C++, and C# projects fall between these extremes, with approximately 95% of dependencies on average specifying version information.

Finding 2. All listed transitive dependencies have version information, but transitive dependencies of unversioned dependencies are omitted. Figure 4.1 also breaks down the analysis per top-level dependency (a.k.a. direct dependencies) and transitive dependencies. We notice that all transitive dependencies include version information across all programming languages. This apparent completeness arises from the behavior of the GitHub SBOM Tool. We examined the generated SBOMs in depth and found that when a parent dependency lacks version information, the tool omits all of its transitive dependencies from the SBOM.

In these cases, only the unversioned parent dependency appears, with none of its downstream dependencies listed. Consequently, the reported 100% version coverage for transitive dependencies does not imply that all transitive dependencies in the projects are versioned; it reflects a conservative approach from the GitHub SBOM Tool, that only includes transitive dependencies of versioned parent dependencies.

Finding 3. There are vast differences in the number of project components across programming languages. Similarly, as reported in previous studies [60], the number of dependencies varies substantially across programming languages. On average, JavaScript and Rust projects exhibit the highest total dependency counts, with 1143 and 558 dependencies per project, respectively,

indicating a strong reliance on third-party libraries. In contrast, Swift and C projects show considerably lower dependency usage, averaging 77.00 and 86.62 dependencies per project.

According to GitHub SBOMs, we observe distinct patterns in how programming languages structure and manage their dependency hierarchies: some ecosystems exhibit broad, direct composition, whereas others rely on deeper chains of indirect dependencies. In particular, Rust, Go, and Java projects tend to include more top-level dependencies on average than other languages, suggesting heavier reliance on direct imports. However, this pattern reverses at the transitive level. On average, JavaScript projects include over 1,024 transitive dependencies, approaching an order-of-magnitude increase relative to their direct imports, and substantially exceeding the transitive dependency counts observed in Rust, Go, and Java projects. This contrast highlights JavaScript projects' exceptionally deep dependency trees, where relatively fewer direct components expand into extensive chains of indirect reuse. Python, PHP, C, C++, C#, and Swift projects exhibit moderate expansion, typically with two to three times more transitive dependencies than top-level components, reflecting layered but more tightly managed dependency structures.

4.1.4 Do GitHub-generated SBOMs provide licensing information for dependencies?

We present a distribution of the percentage of dependencies with licensing information per project, broken down by programming language and dependency type (all, top-level, and transitive) in Figure 4.2. We opt to showcase the distributions rather than averages because, unlike the component version analysis, the data is heavily skewed. Many projects have either 0% or 100% of dependencies licensed, making the mean a poor summary of the underlying variation. Once again, we notice some contrasts related to licensing information across programming languages, and report the following observations:

Finding 1. For projects written in C, C++, Go, PHP, Python, and Swift, license information is absent for the vast majority of dependencies. Figure 4.2 shows that, for C, C++, Go, PHP, and Swift projects, the distribution of license coverage is close to bimodal. More than half of the projects in each of these languages report 0% of their dependencies with license information, while only a

Figure 4.2: The distribution of dependencies' license information per project. Unlike top-level dependencies, not all generated SBOMs include transitive dependencies; thus, the count may not sum up to 1,000 per programming language.

small fraction achieve complete (100%) coverage. Very few projects fall in the intermediate ranges.

Finding 2. SBOMs from JavaScript and Rust projects report license information for a substantially larger share of dependencies. In contrast to other ecosystems, JavaScript and Rust projects demonstrate comparatively high levels of license information reporting. In JavaScript, more than half of the projects provide license information for nearly all dependencies. Rust projects exhibit a similar trend, with nearly one-quarter achieving complete or near-complete license reporting. This observation is consistent with the findings of Wu et al. [61], who analyzed the declared license metadata of 33,710,877 versioned packages across five package management platforms and reported that npm (JavaScript) and Cargo (Rust) exhibit the highest percentages of clearly declared license information among the studied ecosystems. This suggests that the higher license coverage we observe in the SBOMs of JavaScript and Rust projects reflects the stronger license declaration practices of their underlying package ecosystems.

Finding 3. C#, Java, and Python projects have varying degrees of license information. When analyzing projects written in C#, Java, and Python, we notice a more uniform distribution in the proportion of dependencies for which license information is reported. GitHub-generated SBOMs vary in the degree of license information; however, across all ecosystems, fewer than 25% of projects provide license information for all their dependencies. This suggests that most projects will have a sizeable share of components without license information in the GitHub-generated SBOMs for these programming languages.

Finding 4. License information is almost always present in listed transitive dependencies. The differences in license availability across ecosystems are mainly due to how the GitHub SBOM Tool reports licenses for top-level dependencies, as shown in Fig. 4.2. However, for the listed transitive dependencies, license information is almost always included, regardless of the language. As discussed in our previous analysis, GitHub SBOM Tool adopts a conservative approach and only lists transitive dependencies of parent dependencies that have a version.

4.2 RQ2: How consistent are GitHub SBOMs compared to SBOMs from other generators?

While RQ1 focused exclusively on evaluating the completeness of SBOMs generated by GitHub, this research question broadens the scope to assess how GitHub SBOMs compare with SBOMs produced by other widely used generation tools in practice. The goal is to understand whether GitHub SBOMs provide comparable metadata quality, particularly in attributes required for effective vulnerability analysis and license compliance assessment. Accordingly, we analyze SBOMs generated by GitHub, Trivy, Syft, and the Microsoft SBOM Tool for the selected repositories. The procedures used to generate these SBOMs are described in Chapter 3; in this research question, we focus exclusively on analyzing and comparing the contents of the resulting SBOMs.

4.2.1 Approach

Using the SBOMs generated for the selected repositories, we analyze differences at two levels: dependency detection and metadata completeness. We first examine variations in the number of components reported by each tool to capture differences in dependency discovery. We then assess the presence of version and license metadata at the component level, applying consistent validation criteria and aggregating results at the repository level to enable comparison across sources and programming languages.

Component Count Analysis. For all the SBOMs generated by the four tools, we first examined the number of components reported in each SBOM. Due to the highly skewed distribution of component count data across SBOM sources, we employ two statistical methods, as done in previous studies [62, 63]. First, we employed the Kruskal–Wallis H-test [64], a non-parametric test used for comparing multiple independent samples. For each programming language, this test was used to assess whether the SBOMs produced by the four tools differ statistically in the number of reported components across repositories. When the Kruskal–Wallis test indicated significant differences, we conducted Dunn's post-hoc test [65] with Bonferroni correction to examine pairwise differences in component count distributions between GitHub SBOMs and those produced by each of the other

tools.

Validation Criteria. We choose to focus on version and license information to assess the comparative completeness of the SBOMs produced by the four tools. Each metadata attribute was evaluated for presence and validity using the following criteria:

- **Version information** was considered present if an SBOM entry contained a non-empty `versionInfo` or `version` field that was not marked as "UNKNOWN".
- **License information** was considered present if any of `licenseDeclared`, `licenseConcluded`, or `licenseInfoFromFiles` contained meaningful values other than "NOASSERTION", "NONE", or "UNKNOWN".

Percentage Calculation and Aggregation. For each SBOM, we computed the percentage of components that contained valid values for each metadata attribute. For every tool–language combination, these per-SBOM percentages were averaged across all valid SBOM files. This repository-centric aggregation ensures each repository contributes equally to the final results, regardless of its dependency graph size.

4.2.2 Results

We present here the main empirical findings from our analysis, highlighting differences in component detection and metadata reporting across SBOM sources and programming languages.

Finding 1. GitHub SBOMs list significantly more components per project than the SBOMs of all other generation tools, except in Go projects. Figure 4.3 shows clear variation in the number of components detected across the four sources. In many ecosystems, including C#, JavaScript, PHP, and Python, GitHub SBOMs report the highest average number of components. Trivy SBOMs consistently report the fewest components across most languages, while SBOMs of Syft and Microsoft generally fall in the middle.

Finding 2. Go projects show consistency in the component counts across all SBOM sources. In our analysis using the Kruskal–Wallis H-test, we found no statistically significant differences in

the average number of components of Go projects detected across all SBOM sources ($p = 0.186$). Likely due to Golang native support in listing software components, the SBOMs of all four tools show a consistent average number of components per project. In contrast, projects from all other languages exhibit statistically significant differences in detected component counts ($p < 0.05$).

Finding 3. Pairwise comparisons reveal significant differences in component consistency across sources. In the nine programming languages with significant differences, we performed Dunn's post-hoc test with Bonferroni correction to examine pairwise differences between GitHub SBOMs and those of the other tools.

The pairwise analyses revealed the following patterns:

- **GitHub vs Microsoft**: GitHub and Microsoft SBOMs show no consistency in the number of components per project in all the remaining nine languages ($p < 0.05$), indicating significant differences in the reported components between the two sources.
- **GitHub vs Syft**: GitHub and Syft SBOMs showed comparable component count distributions in Java ($p = 0.466$) and Rust ($p = 0.601$). All other languages showed significant differences in the distribution of components.
- **GitHub vs Trivy**: Only Java projects showed consistency in the number of components between the GitHub and Trivy SBOMs ($p = 0.645$). The remaining eight languages showed significant differences in the reported distributions of components.

Surprisingly, only SBOMs generated for Go projects and Java projects showed some level of consistency across sources. Across most of our dataset, GitHub SBOMs show a statistically significant difference in the number of reported components compared to those of Microsoft, Syft, and Trivy.

Finding 4. Microsoft SBOMs consistently provide complete component versioning (100%), while GitHub SBOMs report 94% of versioned components. We present in Figure 4.4 a heatmap of the average versioned component per project. We notice that Microsoft SBOMs provide version information for nearly every component across all languages. GitHub SBOMs follow closely, reporting version data for almost all components except in Java (85%) and Python projects (80%).

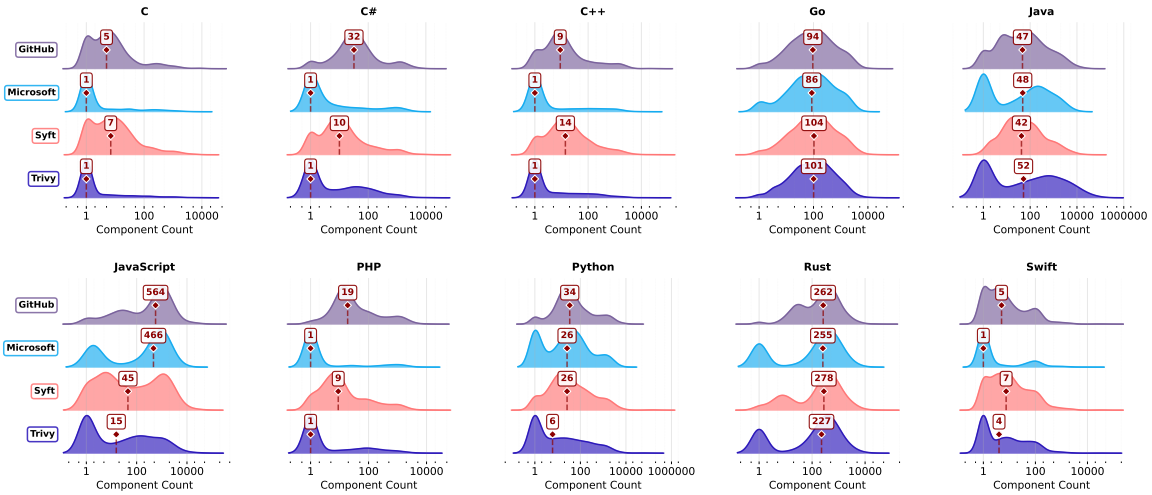

Figure 4.3: Component Count Distribution Per Repository by SBOM Source and Language

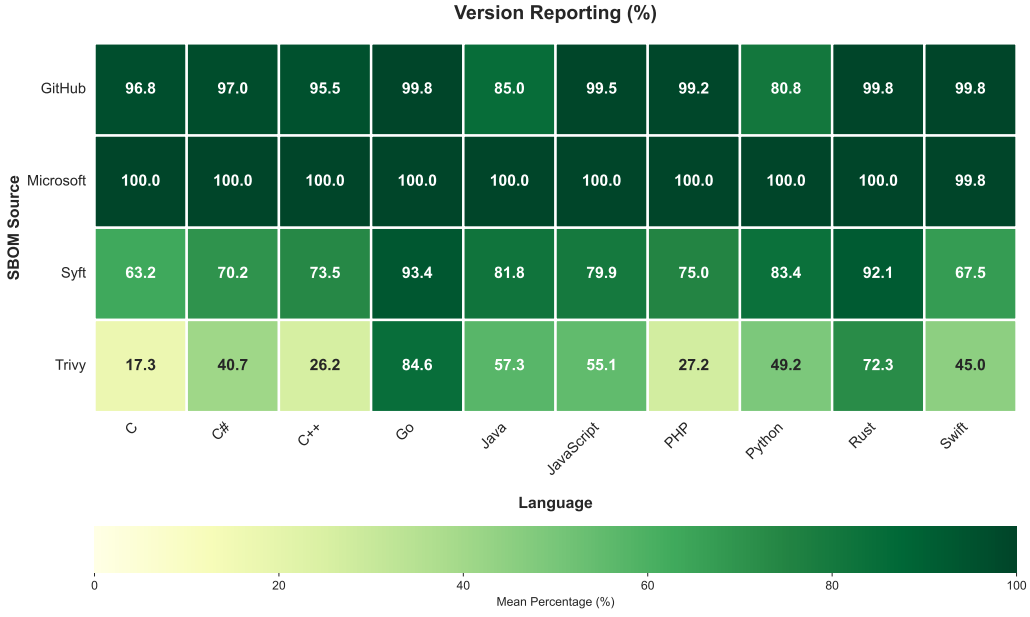

Figure 4.4: Average percentage of versioned components per project across languages and SBOM sources.

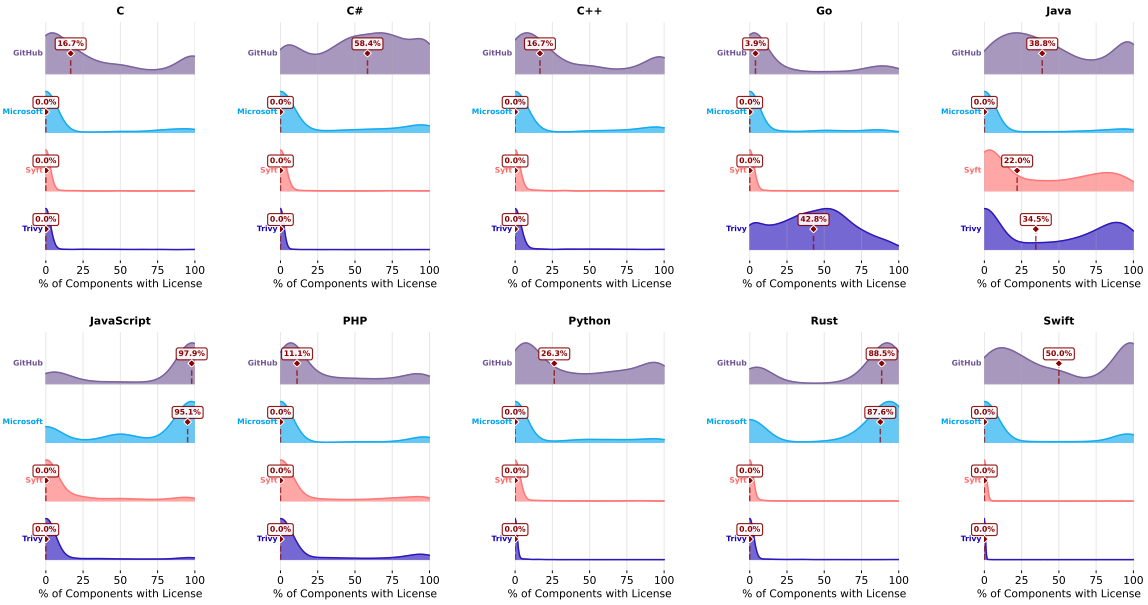

Figure 4.5: Distribution of License Availability Percentage Per Project by SBOM Source and Language

Syft SBOMs perform well for Go and Rust projects, where approximately 90% of components contain version information, but for projects in other languages, their coverage decreases to the 60–80% range. Trivy SBOMs show the weakest coverage overall, never reaching 90% version reporting in any language and in some cases providing version data for only 15–30% of components per project.

Finding 5. GitHub SBOMs provide more complete component license information than those of other tools for most programming languages. We show in Figure 4.5 a distribution of the license availability percentage per project, across all sources and programming languages. We notice that GitHub SBOMs provide more complete license information in nine out of ten languages, showing higher averages. However, even in these cases, coverage remains far from complete: only in Rust and JavaScript projects do GitHub SBOMs achieve an average of more than 80% of components' license information. Microsoft SBOMs deliver comparable coverage to GitHub SBOMs for JavaScript and Rust projects. For Java projects, SBOMs generated by Trivy and Syft report license information at rates similar to GitHub SBOMs. Go projects stand out as a notable exception: Trivy SBOMs substantially outperform all others by a wide margin.

4.3 RQ3: How useful are GitHub SBOMs compared to SBOMs from other generators in vulnerability tracking?

Vulnerability detection is one of the most critical use cases of SBOMs. Modern software ecosystems contain large numbers of third-party libraries, and even a single outdated or vulnerable component can introduce significant security risks [6, 42]. SBOMs are intended to support this process by providing the information necessary to match components against vulnerability databases and evaluate their security status. Hence, in this RQ, we assess how useful GitHub SBOMs are in detecting vulnerabilities in a software product and compare them with the SBOMs of the three other widely used generation tools.

4.3.1 Approach

PURL-Based Vulnerability Querying. To track software components, we rely on Package URLs (PURLs), which provide a canonical and ecosystem-agnostic mechanism for uniquely identifying software components. We use OSV.dev, a distributed vulnerability database for open-source software, as our vulnerability data source [66]. OSV.dev aggregates vulnerability information from multiple authoritative upstream sources and supports vulnerability retrieval through a public API. Vulnerabilities can be queried either by specifying a *component* (or *package*)'s name and ecosystem or by providing a PURL. In our analysis, we exclusively use PURLs to ensure precise and consistent component matching.

We process all SBOMs generated by the four tools and attempt to extract PURLs for every component listed in each SBOM. For each SBOM, we examine the entry corresponding to every component and check for the presence of an `externalRefs` array, search for an item with `referenceType` set to `purl`, and extract the associated `referenceLocator` value, which contains the Package URL (PURL) of the component. After collecting all available PURLs within a given SBOM, we submit them together to the batch query endpoint of the `osv.dev` API. The API returns all known vulnerabilities associated with each queried component, and we aggregate these results to obtain the complete set of vulnerabilities corresponding to the components listed in that SBOM.

Vulnerability Count Analysis. We first analyzed the total number of vulnerabilities associated with each SBOM and examined the distribution of vulnerability counts reported. Similar to the component count analysis (Section 4.2), we used the Kruskal–Wallis H-test and Dunn's post-hoc test with Bonferroni correction to compare the distribution of vulnerability counts across SBOM sources.

4.3.2 Results

Finding 1. PURL reporting is consistent in GitHub and Microsoft SBOMs, while Syft and Trivy SBOMs show inconsistency. GitHub SBOMs provide a PURL for every component across all languages (100% coverage). Microsoft SBOMs perform similarly, reporting PURLs for nearly all listed components, with a negligible drop observed only for Swift (99.8%).

Syft SBOMs report PURLs for a substantially smaller fraction of the listed components, and their coverage varies notably by language. On average, Syft SBOMs report PURLs for approximately 60–90% of the listed components for all the languages. Trivy SBOMs report PURLs for the smallest fraction of listed components overall. For projects written in certain languages, including C (17.7%), C++ (27.1%), and PHP (27.3%), fewer than one-third of the listed components include a PURL. These results indicate that, unlike GitHub and Microsoft SBOMs, those of Syft and especially Trivy frequently omit PURLs for a large portion of the components they list.

Finding 2. PURL availability is highest for Go and Rust projects. Across all sources, Go and Rust projects show the highest PURL availability. Both GitHub and Microsoft SBOMs include PURLs for every component in Go and Rust projects. Syft and Trivy SBOMs, despite their overall inconsistency, also perform relatively well for Go and Rust, achieving their highest PURL coverage in these projects. On average, Syft SBOMs include PURLs for 94.0% of components in Go projects and 91.9% in Rust projects. For Trivy, the values are 89.4% in Go and 70.9% in Rust projects.

Finding 3. The number of vulnerabilities detected varies significantly across sources. As shown in Figure 4.7, GitHub SBOMs consistently map to a higher number of vulnerabilities than the SBOMs of the other tools across almost all programming languages. This outcome was expected, as GitHub SBOMs generally list more components and provide a PURL for each listed component,

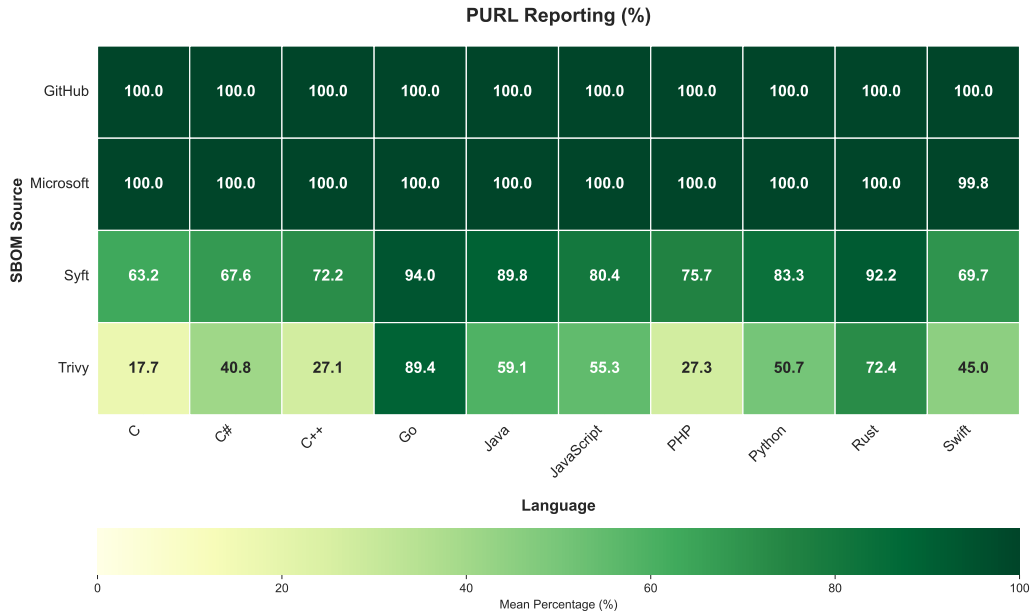

Figure 4.6: Average percentage of components with valid PURLs across languages and SBOM sources.

enabling complete vulnerability lookups via OSV.dev.

Among the remaining tools, the SBOMs of Syft and Microsoft yield broadly similar vulnerability counts, though their relative differences vary by language. Syft SBOMs yield more vulnerabilities in Python, C, and C++ projects, while Microsoft SBOMs yield more in JavaScript and PHP projects. Trivy SBOMs consistently yield the lowest number of vulnerabilities across nearly all languages. This can be attributed to their combination of listing fewer components and frequently omitting PURLs for the components they do report, which limits the effectiveness of OSV.dev-based queries.

Applying the Kruskal–Wallis H-test to the vulnerability counts for each programming language, we found that all ten languages showed statistically significant differences ($p < 0.05$) in vulnerability detection across the four SBOM sources. Dunn's post-hoc test with Bonferroni correction to examine pairwise differences in vulnerability detection between GitHub SBOMs and those of each competing tool revealed the following patterns:

- **GitHub vs Microsoft**: The number of vulnerabilities reported between the two sources was consistent in C# ($p = 0.224$), JavaScript ($p = 0.112$), and Swift projects ($p = 0.421$). All

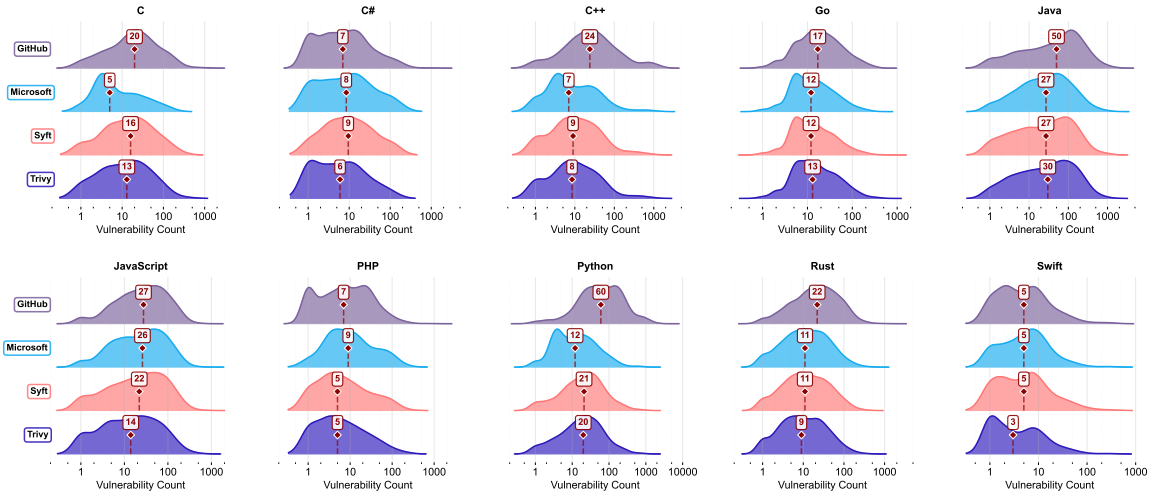

Figure 4.7: Vulnerability Count Distribution Per Repository by SBOM Source and Language.

other languages showed statistically significant differences.

- **GitHub vs Syft**: GitHub and Syft SBOMs showed a consistent number of vulnerabilities only in C ($p = 0.127$), PHP ($p = 0.353$) and Swift projects ($p = 0.944$). Projects from all other languages showed a distinct number of vulnerabilities.

- **GitHub vs Trivy**: GitHub and Trivy SBOMs showed a consistent number of vulnerabilities in projects written in C# ($p = 0.224$) and PHP ($p = 0.052$). The two sources diverge significantly in all other remaining languages.

While no single programming language showed consistent vulnerability counts across all sources, GitHub SBOMs are sporadically consistent with those of other tools in specific languages, usually related to C#, PHP and Swift.

A caveat about the number of vulnerabilities. The precision of vulnerability detection depends on the availability of version information. Our analysis indicates that while OSV queries based on PURLs always return vulnerability information for a given dependency, the precision of these results critically depends on whether the PURL specifies an exact version. If a GitHub SBOM does not include an exact version for a component, the resulting PURL is versionless. Figure 4.8 illustrates six different ways the component `setuptools` appears across the SBOMs. It can be observed that only when an exact version—`77.0.3` in this case—is provided does the PURL reflect that version;

Figure 4.8: Effect of presence of version information on PURL.

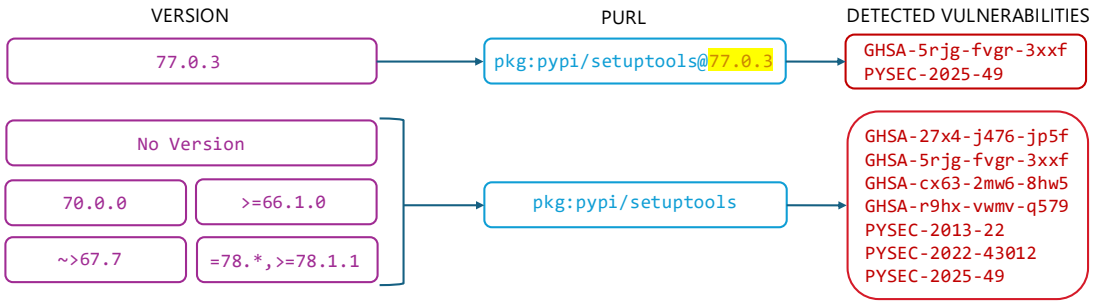

Figure 4.9: How version information's availability in PURL affects detection of vulnerabilities.

in all other cases, where no version or a non-exact version is provided, the PURL is versionless.

Submitting a versionless PURL to the `osv.dev` API returns all vulnerabilities associated with that component, regardless of version. For example, as shown in Figure 4.9, querying with the PURL `pkg:pypi/setuptools@77.0.3` returns two vulnerabilities, enabling the software consumer to identify exactly which vulnerabilities affect their software. In contrast, querying with the versionless PURL `pkg:pypi/setuptools` returns six vulnerabilities, leaving the consumer uncertain about which vulnerabilities are relevant to their software.

Chapter 5

Discussion

In this chapter, we discuss four different aspects of our results. We first present an accuracy evaluation of the four SBOM tools against a manually constructed ground truth for Python, Java, and JavaScript (Section 5.1). We then compare the component agreement level between the GitHub SBOM Tool and the Microsoft SBOM Tool (Section 5.2), discuss the lack of component versioning in Python projects (Section 5.3), and examine the lack of license information across ecosystems (Section 5.4).

5.1 Accuracy Evaluation Against Ground Truth

The analyses described in RQ2 (Chapter 4, Section 4.2) measure the *presence* of metadata fields in generated SBOMs, but do not assess whether the reported values are *correct*. A tool may report an incorrect version or omit the license, undermining further compliance analysis, or fail to report components that the repository actually uses, introducing false negatives that leave real dependencies invisible to downstream consumers.

To address this, we construct a ground truth dataset for Python, Java, and JavaScript, three languages whose package ecosystems provide programmatic access to authoritative dependency, version, and license metadata, enabling reliable automated ground truth. Given the time-consuming task of building projects to resolve their actual dependency tree, we randomly sampled 200 repositories per language from our full dataset, yielding 600 repositories in total.

For constructing the ground truth, we resolve each repository's dependencies programmatically from each ecosystem's native tooling.

- For Python, following the methodology of Yu et al. [28], we install each repository's dependencies into a fresh virtual environment using `pip install` and then capture the full resolved environment via `pip freeze`. This produces the complete set of transitive dependencies, each annotated with its exact pinned version.
- For Java, we run `mvn dependency:tree` against each repository's `pom.xml`, which triggers Maven's dependency resolver and returns the fully resolved transitive dependency tree, including the concrete version selected for each artifact.
- For JavaScript, we parse `package-lock.json` or `yarn.lock`, whichever is present in the repository, to extract the full set of installed packages together with their locked versions. During repository sampling, only repositories containing at least one of these two lockfiles were selected; repositories lacking both lockfiles were excluded and replaced by the next sampled candidate.

For license ground truth, we query each ecosystem's authoritative registry: the PyPI [67] API for Python; the npm [68] registry API for JavaScript; and the Libraries.io [69] API for Java, since Maven Central does not expose a structured REST API for license data and its POM `<licenses>` field is free-form text that is unsuitable for automated normalisation.

We evaluate tools at the **component detection** level (Precision, Recall, F1 via normalized name matching) and the **version accuracy** and **license accuracy** levels (fraction of True Positives where the reported value exactly matches the ground truth, macro-averaged across repositories). For version accuracy, we require an exact match between the version reported in the SBOM and the version present in the ground truth. A tool that reports a version range or a resolved value that differs from the ground truth is considered incorrect.

Table 5.1 presents the component detection, version accuracy, and license accuracy results evaluated against our ground truth dataset across Python, Java, and JavaScript. While RQ2 consistently positioned the GitHub SBOM Tool as the leading tool in terms of component count and metadata

Table 5.1: Component detection, version reporting accuracy, and license reporting accuracy evaluated against ground truth, macro-averaged across repositories. Best value per metric per language is shown in **bold**.

Language	Tool	Component Detection Precision	Component Detection Recall	Component Detection F1	Version Reporting Accuracy	License Reporting Accuracy
Python	GitHub SBOM Tool	33.7%	57.3%	29.0%	21.6%	19.7%
	Microsoft SBOM Tool	**44.4%**	**66.4%**	**47.9%**	**69.8%**	**27.6%**
	Trivy	20.3%	32.3%	19.8%	27.4%	1.7%
	Syft	13.2%	34.1%	13.2%	31.0%	3.9%
Java	GitHub SBOM Tool	46.8%	38.2%	35.9%	53.7%	28.5%
	Microsoft SBOM Tool	**79.3%**	**98.4%**	**84.6%**	**95.6%**	8.7%
	Trivy	55.7%	54.9%	50.5%	81.9%	**60.2%**
	Syft	55.7%	36.9%	38.7%	73.5%	1.3%
JavaScript	GitHub SBOM Tool	**88.2%**	**86.7%**	**85.3%**	**90.4%**	**94.7%**
	Microsoft SBOM Tool	87.9%	80.3%	81.1%	88.8%	78.5%
	Trivy	65.1%	26.5%	30.1%	66.9%	1.3%
	Syft	63.8%	40.0%	41.2%	69.2%	25.6%

presence, the accuracy evaluation reveals a more nuanced picture. For component detection, the GitHub SBOM Tool's higher component count in Python and Java does not translate into higher accuracy against the ground truth; the Microsoft SBOM Tool, despite reporting fewer components, achieves better precision, recall, and F1 in both ecosystems. For version accuracy, the Microsoft SBOM Tool's near-complete version presence observed in RQ2 does carry over into correctness: it leads on version accuracy in two of the three ecosystems, confirming that it not only populates version fields consistently but also reports the right values. For license accuracy, three different tools lead in three ecosystems.

In **Python**, the Microsoft SBOM Tool leads on all component detection metrics, achieving a Precision of 44.4%, Recall of 66.4%, and F1 of 47.9%, alongside the highest version accuracy of 69.8% and license accuracy of 27.6%. This is somewhat surprising given that RQ2's Finding 1 (Section 4.2.2) identified the GitHub SBOM Tool as detecting more components than the Microsoft SBOM Tool in Python in terms of raw component counts. However, a higher component count does not imply higher accuracy: the GitHub SBOM Tool surfaces more components overall, but fewer of them match the ground truth, resulting in lower component detection performance than the Microsoft SBOM Tool. Version accuracy is low across all tools except the Microsoft SBOM Tool, ranging from 21.6% (GitHub SBOM Tool) to 31.0% (Syft). The Microsoft SBOM Tool's relatively

high version accuracy of 69.8% suggests that it resolves undeclared or range-based versions more reliably than other tools while the other tools may provide range-based versions or wrong resolved value. License accuracy is uniformly poor across all tools, with no tool exceeding 27.6%.

In **Java**, the Microsoft SBOM Tool dominates component detection with a Precision of 79.3%, Recall of 98.4%, and F1 of 84.6%, and achieves the highest version accuracy at 95.6%. RQ2's Finding 4 showed that Microsoft SBOM Tool provides version information for every detected component across nearly all ecosystems, and the 95.6% version accuracy confirms that these reported versions are also almost always correct for Java projects. The GitHub SBOM Tool performs considerably weaker in component detection (F1 of 35.9%) and version accuracy (53.7%), consistent with our earlier observation in RQ1's Finding 2 (Section 4.1) that GitHub SBOMs omit transitive dependencies of unversioned parent components. Notably, the Microsoft SBOM Tool's dominant component detection performance does not extend to license accuracy, where it achieves only 8.7% compared to GitHub SBOM Tool's 28.5% or Trivy's 60.2%.

In **JavaScript**, the GitHub SBOM Tool leads across all metrics, achieving a Precision of 88.2%, Recall of 86.7%, F1 of 85.3% for component detection, version accuracy of 90.4%, and license accuracy of 94.7%. The strong version accuracy of the GitHub SBOM Tool in JavaScript suggests that it benefits from the availability of lockfiles, which provide exact pinned versions and remove the need to resolve range-based declarations. The Microsoft SBOM Tool is competitive across component detection and version accuracy (81.1% and 88.8% respectively) but falls more behind on license accuracy (78.5%). The license accuracy results for both tools are consistent with RQ2's Finding 5 (Section 4.2.2), which identified JavaScript as one of only two languages where the GitHub SBOM Tool achieves above 80% license presence and Microsoft SBOM Tool also reports similar performance for JavaScript projects. Our ground truth analysis confirms that the reported licenses are not only present but also largely correct. Trivy and Syft show sharply lower performance in component detection, particularly on recall (26.5% and 40.0%), confirming that they detect a selective subset of the full dependency tree in this ecosystem.

5.2 GitHub vs Microsoft SBOM Generation tool

Among the SBOM generation tools evaluated, both the Microsoft SBOM Tool and the GitHub SBOM Tool demonstrated reliable performance in providing component version and license information (RQ2) and PURL information (RQ3). However, high completeness scores do not preclude the possibility that the two tools are detecting substantially different sets of dependencies.

To examine whether the tools also agree on the components they report, we perform a component-level agreement analysis. For the component-level comparison, each component's PURL is normalized by removing the version from the PURL, causing multiple versions of the same component to collapse into a single entry; we refer to these as *unique components*. For the versioned comparison, each entry is keyed by the pair (versionless PURL, `versionInfo`), such that the same component appearing with different versions is treated as distinct entries; we refer to these as *unique versioned components*. Packages lacking version information are excluded from the versioned comparison.

Table 5.2 shows the percentage of components that are uniquely reported in GitHub SBOMs (GH), reported in both GitHub and Microsoft SBOMs (GH $\cap$ MS), and uniquely reported to Microsoft (MS). In this analysis, we first summed all three component categories per project and computed the percentage of the aggregated sum across all unique components found in both tools' SBOMs.

Overall, in most languages GitHub SBOM Tool and Microsoft SBOM Tool tend to agree on the majority of unique components. The highest agreement is observed in Go (76.8%), JavaScript (75.0%), and Rust (72.8%) projects, suggesting strong consistency. At the same time, GitHub tends to identify more components in 8 out of 10 languages, with Java and Rust projects being the only exceptions. In contrast, PHP exhibits the lowest overlap (43.0%), where more than half of the components (50.5%) are reported exclusively by GitHub, indicating substantial divergence between the tools.

When accounting for versioned components, i.e., whether both GitHub SBOM Tool and Microsoft SBOM Tool report the same component and their versions, the agreement decreases. For example, in Go projects, agreement drops from 76.8% (components) to 32.6% (versioned components), with both tools reporting roughly one-third of versioned components exclusively. A similar

Table 5.2: Agreement level between GitHub SBOM Tool (GH) and the Microsoft SBOM Tool (MS). We highlight in light grey values above 25% and in dark grey values above 50%.

Languages	Agreement Level					
	Unique Components			Unique Versioned Components		
	GH	GH∩MS	MS	GH	GH∩MS	MS
C	28.3%	52.5%	19.1%	33.4%	43.9%	22.8%
C#	38.2%	52.7%	9.1%	42.4%	43.3%	14.3%
C++	27.5%	53.7%	18.8%	33.4%	41.7%	24.9%
Go	13.0%	76.8%	10.3%	34.0%	32.6%	33.4%
Java	17.6%	49.2%	33.1%	22.8%	36.1%	41.1%
JavaScript	15.1%	75.0%	9.9%	22.4%	62.4%	15.2%
PHP	50.5%	43.0%	6.5%	51.8%	36.4%	11.8%
Python	24.2%	57.9%	17.9%	29.6%	45.9%	24.5%
Rust	13.2%	72.8%	14.0%	29.1%	44.6%	26.3%
Swift	20.5%	66.1%	13.3%	23.2%	60.0%	16.8%

decline is observed in Java and C++ projects, suggesting that version resolution and normalization strategies differ significantly. Java, in particular, shows a notable asymmetry, with Microsoft reporting 41.1% of versioned components exclusively compared to 22.8% for GitHub. These results indicate that while the tools often agree on the presence of dependencies at the component level, discrepancies become more pronounced when version information is incorporated.

5.3 Potential causes for lack of versioning of Python dependency components

From the results of RQ2, we observe that GitHub SBOM Tool provides low support for component versioning in Python projects, with an average of 80% of components per project being versioned.

To understand the potential causes, we automatically analyzed the `requirements.txt` files of all Python projects in our dataset. Python allows developers to import and use a library without specifying it in any metadata file, as long as it is installed in the runtime environment. Furthermore, Python projects offer multiple mechanisms for specifying dependencies, including `requirements.txt`, `setup.py`, `pyproject.toml`, and `Pipfile`,

among others — though our analysis focused on `requirements.txt` files, which are the most widely used convention for specifying installation requirements. Among the 9,575 PyPI library instances missing version information in GitHub SBOM Tool-generated SBOMs, we identified distinct causes.

- **Lack of version specifiers.** In 5,961 cases (62.26%), the libraries were present in requirements.txt files but declared without any version specifier, which directly propagates to the SBOM as an unversioned entry.
- **Absent libraries in the manifest file.** In the remaining 3,614 cases (37.74%), the libraries were absent from requirements.txt entirely, attribu-table to one of three scenarios: the dependency may be declared in an alternative metadata file not covered by our analysis; it may be a transitive dependency resolved by GitHub from the full dependency tree; or it may not be declared anywhere at all, with GitHub SBOM Tool surfacing it in the SBOM through static analysis of import statements in the source code.

These findings suggest that the version-adherence problem in GitHub SBOMs is primarily due to loose dependency-declaration practices in the Python ecosystem rather than a limitation of the SBOM generation tool itself.

5.4 Why is licensing information so seldom reported in SBOMs?

Our results show that component license information is seldom reported in SBOMs, across all tools. This may indicate that poor license coverage is not just a tool limitation, but likely reflects a limitation on how license metadata is resolved. We break the analysis of the license into two orthogonal factors: 1) dependency versioning practices and 2) the related package manager ecosystem. To this aim, we classify every dependency exported across all SBOMs analyzed based on their reported version into three categories:

- **Exact version.** We classify a dependency as having an exact version if the version is expressed in the SBOM as a unique version, without ranges or semantic versioning, a.k.a., dependency pinning.

Table 5.3: License Availability Across Package Managers Based on Version Specification

Package Manager	Total Dependencies	Exact Version Dependencies	Exact Version Licensed %	Non-Exact Version Dependencies	Non-Exact Version Licensed %	No Version Dependencies	No Version Licensed %
pypi	82,191	54,647	86.20%	12,679	0.00%	14,865	0.00%
cargo	353,825	319,231	98.71%	34,594	0.00%	0	0.00%
githubactions	58,953	58,906	0.00%	47	0.00%	0	0.00%
maven	114,587	74,150	48.21%	226	0.00%	40,211	0.00%
nuget	43,206	31,809	66.56%	9,556	0.00%	1,841	0.00%
npm	2,708,803	2,647,719	99.18%	61,070	0.00%	14	0.00%
golang	128,770	128,770	0.00%	0	0.00%	0	0.00%
gem	25,048	24,258	97.34%	790	0.00%	0	0.00%
composer	41,970	30,041	41.76%	11,501	0.00%	428	0.00%
pub	3,333	2,872	0.00%	461	0.00%	0	0.00%
swift	3,854	3,854	0.00%	0	0.00%	0	0.00%
github	208	208	0.00%	0	0.00%	0	0.00%
deb	589	589	0.00%	0	0.00%	0	0.00%

- **Non-exact version**. We classify a dependency's version as *Non-exact* if it is expressed as a range, a wildcard, or using semantic versioning (semver) operators.
- **No version**. Dependencies where no version is specified in the SBOM.

Table 5.3 presents license availability across package managers, broken down by component version strategy [42]. We observe a clear pattern across all package managers: dependencies reported with an exact version are far more likely to include license information, whereas those with missing or non-exact versions consistently lack versions. We also observe that exact versioning alone does not guarantee license availability. For dependencies managed by the package managers `swift`, `pub`, `deb`, `golang`, `github`, and `githubactions`, the tool fails to retrieve license information for all dependencies, regardless of the versioning strategy. Based on this, we examined all unlicensed dependencies per language and attributed each to one of the following probable causes:

- **Explanation-1 (E1):** Dependency is found in the SBOM having a non-exact version or having no version, e.g., dependencies declared using semantic versioning.
- **Explanation-2 (E2):** Dependency is listed in the SBOM with a `pkg:githubactions` type, indicating it is a GitHub Action referenced within the project's CI/CD pipeline.

Table 5.4: Explanations for lack of license information

Language	# Unlicensed	Non-Exact/ No Version (E1)	GH Actions (E2)	Golang (E3)	Pub (E4)	Swift (E5)	GitHub (E6)	Deb (E7)	Unaccounted
C	11360	3101 (27.3%)	4779 (42.1%)	1869 (16.5%)	637 (5.6%)	15 (0.1%)	0 (0.0%)	0 (0.0%)	959 (8.4%)
C#	30050	13953 (46.4%)	4660 (15.5%)	182 (0.6%)	92 (0.3%)	13 (0.0%)	16 (0.1%)	0 (0.0%)	11134 (37.1%)
C++	19344	7891 (40.8%)	6584 (34.0%)	1728 (8.9%)	412 (2.1%)	72 (0.4%)	191 (1.0%)	220 (1.1%)	2246 (11.6%)
Go	135288	4135 (3.1%)	9713 (7.2%)	119099 (88.0%)	140 (0.1%)	8 (0.0%)	0 (0.0%)	0 (0.0%)	2193 (1.6%)
Java	96736	47195 (48.8%)	5805 (6.0%)	1868 (1.9%)	1022 (1.1%)	26 (0.0%)	1 (0.0%)	0 (0.0%)	40819 (42.2%)
JavaScript	44342	28766 (64.9%)	4219 (9.5%)	1573 (3.5%)	0 (0.0%)	16 (0.0%)	0 (0.0%)	0 (0.0%)	9768 (22.0%)
PHP	38607	14569 (37.7%)	4582 (11.9%)	485 (1.3%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	18971 (49.1%)
Python	40260	23634 (58.7%)	7423 (18.4%)	988 (2.5%)	337 (0.8%)	2 (0.0%)	0 (0.0%)	369 (0.9%)	7507 (18.6%)
Rust	60916	44691 (73.4%)	8909 (14.6%)	899 (1.5%)	18 (0.0%)	32 (0.1%)	0 (0.0%)	0 (0.0%)	6367 (10.5%)
Swift	7025	348 (5.0%)	2232 (31.8%)	79 (1.1%)	214 (3.0%)	3670 (52.2%)	0 (0.0%)	0 (0.0%)	482 (6.9%)

- **Explanation-3 (E3):** Dependency appears in the SBOM with a `pkg:golang` type, indicating it is a Golang component.
- **Explanation-4 (E4):** Dependency is listed in the SBOM with a `pkg:pub` type, indicating it is a Pub component.
- **Explanation-5 (E5):** Dependency is found in the SBOM with a `pkg:swift` type, indicating it is a Swift component.
- **Explanation-6 (E6):** Dependency appears in the SBOM with a `pkg:github` type, indicating it is a component sourced directly from a GitHub repository rather than a formal registry.
- **Explanation-7 (E7):** Dependency is listed in the SBOM with a `pkg:deb` type, indicating it is a Debian system-level component.

Table 5.4 presents the result of our analysis across all ten languages. Whenever possible, each dependency was categorized in one of the explanations. Dependencies that could not be attributed to any of the identified explanations are reported as *Unaccounted*.

The most pervasive cause of missing license information is a non-exact or absent version specification (E1). In 8 out of 10 languages, E1 is the single largest contributor to unlicensed dependencies. Among dependencies that do have an exact version, missing licenses are explained by membership in package managers for which the GitHub SBOM Tool does not retrieve license information (E2–E7). The degree to which these causes account for the remaining missing licenses varies substantially across languages. For Go, Golang dependencies (E3) alone account for 88.0% of all unlicensed dependencies, leaving only 1.6% unaccounted for. Swift is similarly well-explained,

with Swift dependencies (E5) and GitHub Actions (E2) together covering 84.0% of missing licenses. In contrast, Java and PHP prove harder to explain: even after accounting for all identified causes, 42.2% and 49.1% of their unlicensed dependencies, respectively, remain unaccounted for — meaning these dependencies carry exact version information and belong to supported package managers, yet the GitHub SBOM Tool still fails to retrieve their license metadata. This points to gaps in the upstream registries the tool relies upon, and represents an important avenue for future improvement.

5.5 Implications

The completeness and consistency gaps reported throughout this study have different consequences depending on who is consuming the SBOM and for what purpose. We discuss implications for practitioners using SBOMs in security, licensing, and compliance workflows, and for researchers building or evaluating tooling on SBOM data.

5.5.1 Implications for Practitioners

Whether the GitHub SBOM Tool is a good choice depends on the metadata a practitioner needs and the ecosystem of the target project; no single tool dominates across every metric we measured.

Default choice for broad discovery and vulnerability screening. For teams whose primary goal is broad dependency discovery and vulnerability screening, the GitHub SBOM Tool is a reasonable default. It reports a PURL for 100% of listed components in every language (RQ3, Finding 1, Section 4.3), a strict prerequisite for automated OSV.dev-style vulnerability matching, and, except in Go, it lists more components than the other three tools (RQ2, Finding 1, Section 4.2.2). In ecosystems with strict pinning conventions, this breadth also comes with reliable version data: average version coverage exceeds 94% in every language except Java (83%) and Python (78%), and surpasses 99.5% specifically in Go, Rust, and Swift (RQ1, Section 4.1). For JavaScript specifically, our ground-truth evaluation confirms that this reported metadata is not just present but largely correct, with an F1 of 85.3% for component detection and a version accuracy of 90.4% (Section 5.1).

Cross-validation needed for Java and Python. The same recommendation does not hold for

Java and Python. In both ecosystems, the GitHub SBOM Tool's higher raw component counts do not translate into higher accuracy: the Microsoft SBOM Tool detects fewer components overall but achieves markedly higher precision, recall, and version accuracy against ground truth (Section 5.1). Practitioners performing security-sensitive tasks in these two ecosystems, e.g., pinning exact vulnerable versions for remediation, should not rely on the GitHub SBOM Tool's SBOM alone and should cross-validate against the Microsoft SBOM Tool or against ecosystem-native resolution (`pip freeze`, `mvn dependency:tree`).

Vulnerability counts as upper bounds. Practitioners building vulnerability triage on top of the GitHub SBOM Tool's SBOMs should also account for the precision caveat discussed in RQ3 (Section 4.3): a component without an exact version yields a versionless PURL, and querying OSV.dev with a versionless PURL returns every known vulnerability for that component regardless of the version actually in use. Since Java and Python are precisely the ecosystems where the GitHub SBOM Tool most often omits exact versions, vulnerability counts derived from their SBOMs should be treated as upper bounds requiring manual triage, not as an authoritative count of exploitable issues.

Ecosystem-specific tool choice for license auditing. For license-compliance workflows, the GitHub SBOM Tool should not be used as the sole source of truth in any of the ten ecosystems we studied. Even in JavaScript and Rust, its two best-performing ecosystems, average license coverage per project only just exceeds 80% (RQ2, Finding 5, Section 4.2.2), and for five languages (C, C++, Go, PHP, and Swift) the majority of projects report 0% licensed dependencies (RQ1, Section 4.1). Tool choice for license auditing is also ecosystem-specific: in Java, Trivy achieves substantially higher license accuracy than the GitHub SBOM Tool (60.2% vs. 28.5%, Section 5.1) despite detecting far fewer components overall. A plausible explanation is that we ran Trivy with `--license-full` enabled (Section 3.3), which scans the full contents of each dependency's files for embedded license expressions rather than relying only on package metadata; our study did not isolate this factor directly, but it is consistent with Trivy's advantage being concentrated in Java rather than uniform across ecosystems. This gap is nonetheless a useful practical signal: a full-file license scanner such as Trivy can be a valuable complement, not a replacement, for a dependency-graph-based tool such as the GitHub SBOM Tool when license accuracy is the priority, an observation that also motivates the hybrid, per-field tool selection we propose as future work (Section 7.2).

Supplier enrichment for NTIA compliance. Organizations that must submit NTIA-compliant SBOMs, e.g., for U.S. federal procurement, cannot use the GitHub SBOM Tool's output as-is: supplier information is absent from all 10,000 SBOMs we generated, regardless of language (RQ1, Finding 1, Section 4.1). Because the GitHub SBOM Tool derives SBOMs purely from the repository's dependency graph, which does not track publisher identities, achieving compliance requires an additional enrichment step, e.g., resolving the supplier from the package registry associated with each component's PURL, before the SBOM can satisfy the NTIA minimum elements.

5.5.2 Implications for Researchers

Our results also carry implications for how empirical software engineering research treats SBOMs as a data source.

Presence is not a proxy for correctness. RQ2 (Section 4.2.2) measured whether version and license fields were populated, and by that measure the GitHub SBOM Tool appeared to be the strongest performer in most ecosystems. Yet the ground-truth evaluation in Section 5.1 shows that in Python and Java, the Microsoft SBOM Tool, despite reporting fewer components, is more accurate. This gap between presence and correctness suggests that future tool evaluations should, whenever feasible, pair a completeness check with an accuracy check against a resolved ground truth, rather than relying on presence-based comparisons alone.

SBOM generators are not interchangeable. This matters for any study that treats "an SBOM" for a repository as an unambiguous record of its dependency state. Our component-agreement analysis (Section 5.2) shows that even between the two most complete tools, the GitHub SBOM Tool and the Microsoft SBOM Tool, agreement on the versioned identity of a component can drop as low as 32.6% in Go once version is taken into account, despite substantial overlap when version is ignored. A vulnerability or license study that draws conclusions from a single generator's SBOM without reporting which generator was used may not replicate if repeated with a different tool on the same repository snapshot. We recommend that empirical studies using SBOM data explicitly report the generator used and treat generator choice as a methodological variable, in the same way compiler or static-analysis-tool choice is reported elsewhere in software engineering research.

Transitive-dependency truncation can bias large-scale datasets. The conservative truncation behavior we identified in the GitHub SBOM Tool, where transitive dependencies of an unversioned parent are silently omitted rather than listed without a version (RQ1, Finding 2, Section 4.1), produces a systematic, language-dependent undercount that is not visible from the SBOM alone. Researchers building large-scale datasets from GitHub SBOMs, for instance to estimate vulnerability exposure across open-source projects, risk silently underestimating the dependency trees of Java and Python projects specifically, since these are the ecosystems where top-level components most often lack a version (RQ1, Finding 3, Section 4.1). This bias would not surface as missing data but as an artificially small, seemingly complete dependency graph, and should be accounted for when GitHub SBOMs are used as the sole data source for such studies.

The root cause of residual license gaps remains open. Our root-cause analysis of missing license information (Section 5.4) leaves a research-relevant question open rather than closed. For most languages, missing licenses are attributable to non-exact versioning (E1) or to package-manager types the GitHub SBOM Tool does not query for license data (E2–E7). In Java and PHP, however, 42.2% and 49.1% of unlicensed dependencies, respectively, remain unaccounted for despite carrying an exact version and belonging to a supported package manager. Because our analysis observes only the SBOM output, it cannot on its own determine whether this residual gap originates in the GitHub SBOM Tool's queries or in the upstream registries it queries (Maven Central, Packagist); we treat this as an open question for the tracing study we propose in Chapter 7 (Section 7.2), rather than as a settled explanation.

Chapter 6

Threats to Validity

In this chapter, we discuss threats to the validity of our study and how we mitigated them.

6.1 Generalization of the Results

Our study's findings may not generalize to all software projects or SBOM tools. We generated 10,000 SBOMs across 10 programming languages using the GitHub SBOM Tool and the three other popular tools: the Microsoft SBOM Tool, Syft, and Trivy. While this covers a variety of languages and tooling approaches, it still represents only a subset of the broader software ecosystem. Projects outside our selection—especially those in different domains or using different dependency management practices—might show different patterns.

Moreover, we only considered open-source projects hosted on GitHub. Proprietary software or projects hosted elsewhere may have different dependency characteristics, so caution is needed when extending our conclusions to those contexts. Future studies including a wider range of languages, tools, and project types are needed to confirm whether our observations hold more generally.

A related limitation concerns our sampling criterion. We select the top 1,000 most-starred repositories per language (Section 3.1); star count is a validated proxy for popularity [48], but not for a project's *criticality*, i.e., how much impact a vulnerability or licensing error in it would have downstream. A popular tutorial repository and a popular cryptographic library are sampled identically,

and we did not label projects by criticality, domain, or industry sector. Our findings therefore describe SBOM quality for popular open-source projects in general; whether they also hold for high-criticality software specifically, where SBOM accuracy arguably matters most, remains an open question that our sampling design cannot answer.

A second limitation concerns the temporal scope of our study: all SBOMs were generated from a single snapshot of each repository (Section 3.3). SBOM maturity is unlikely to be a fixed property of a project or ecosystem; it plausibly depends on a project's domain and use and evolves over time. A single snapshot cannot distinguish a stable ecosystem property from a transient adoption state, so the cross-ecosystem differences we report should be read as a snapshot of current practice rather than a durable ceiling or floor. A longitudinal study sampling the same repositories over time, ideally stratified by domain or criticality, would help test this (Section 7.2).

6.2 Internal Threats to Validity

Potential mismatches in SBOMs could arise if different tools analyze different snapshots or branches of a repository, or if their internal generation mechanisms differ. To control for this, Microsoft SBOM Tool, Trivy, and Syft were executed on locally cloned repositories, using a snapshot of each repository's default branch to ensure identical source code, dependency manifests, and project structures. In contrast, GitHub SBOMs were obtained via the GitHub REST API; although these are also generated from the default branch, the internal mechanisms remain opaque and may differ from those of the other tools. It is also important to note that the Microsoft SBOM Tool did not successfully generate SBOMs for 254 out of 10,000 projects (2.54%). These failures were distributed across multiple programming languages, indicating that they were not concentrated in a single ecosystem. While this represents a small proportion of the overall dataset, the excluded repositories may slightly affect the completeness of the comparison. Nevertheless, the large remaining sample provides broad coverage across languages and helps mitigate the impact of these isolated tool limitations.

6.3 Reproducibility Limitations

Reproducibility in our study differs across tools due to the nature of how SBOMs are generated. The GitHub REST API does not support generating SBOMs for historical states of a repository, making it impossible to recreate the same GitHub-generated SBOMs at a later time. However, our replication package includes all GitHub-generated SBOMs, which can be used directly for reproducibility purposes. In contrast, for Microsoft SBOM Tool, Trivy, and Syft, SBOMs can be regenerated from the same locally cloned repository snapshot, ensuring consistent results across repeated analyses. Each generated SBOM records its creation timestamp, further supporting reproducibility for these three tools.

6.4 Conclusion Validity

Throughout this study, our research questions are framed primarily around programming languages, as we collected repositories labeled under ten different languages on GitHub. However, as discussed in Section 5.4, real-world projects are rarely confined to a single language: most repositories are polyglot in nature, comprising components written in multiple languages and managed by different package managers [70, 71]. This means that the SBOMs we analyze for a given language may contain packages drawn from entirely different ecosystems, and the results we attribute to a language may in part reflect the behavior of those other ecosystems rather than the primary one. Practitioners and researchers should therefore interpret our per-language conclusions with this caveat in mind, as they represent the experience of repositories *labeled* as a given language, not of purely single-language projects.

A related confound is that, because we did not collect independent domain or criticality labels (Section 6.1), we cannot rule out that language is also acting as a proxy for domain. For example, Go and Rust are disproportionately used for infrastructure and systems tooling within our sample, whereas Swift is disproportionately used for end-user mobile applications. The per-language differences we attribute to ecosystem-specific tooling and packaging conventions (Chapters 4 and 5) may therefore partly reflect differences in the typical domain of projects written in each language, rather than purely technical differences between package managers. Disentangling the two would

require sampling stratified by an independently assigned domain or criticality label, in addition to language.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

This study examined the quality of automatically generated SBOMs at scale, using GitHub's native SBOMs as a representative case and comparing them with widely used third-party tools. Our results show that while automated SBOM generation has become easier with the availability of multiple tools, important gaps remain that limit its effectiveness for security and compliance-driven use cases.

In particular, the absence of supplier information, the incomplete and uneven version reporting introduces blind spots that propagate to transitive dependencies and directly weaken vulnerability and license analysis. Our findings demonstrate that missing or non-exact versions force downstream tools to either over-approximate vulnerabilities or ignore affected components entirely.

The comparison across SBOM tools further indicates that no single approach currently offers comprehensive coverage across languages and metadata dimensions. GitHub's SBOMs provide broad dependency visibility and strong identifier support, while the Microsoft SBOM Tool prioritizes version completeness at the cost of coverage. The variability observed across ecosystems suggests that SBOM quality is shaped as much by language-specific tooling and conventions as by the SBOM generator itself.

Overall, these results suggest that automated SBOMs are best viewed as a baseline rather than a complete solution. Improving supplier attribution, version propagation, and license extraction will

be critical for SBOMs to fully support large-scale vulnerability management and supply-chain risk assessment. Our findings highlight concrete areas where SBOM tooling and standards can evolve to better align with their intended role in securing modern software supply chains.

7.2 Future Work

The strengths and limitations identified in this study suggest several promising directions for future research.

Extending the ground truth evaluation to more ecosystems. Our accuracy evaluation against ground truth (Section 5.1) was limited to Python, Java, and JavaScript, the three ecosystems for which authoritative dependency resolution and registry metadata could be obtained programmatically at scale. The near-total absence of license information for `golang`, `swift`, and `pub` packages, were observed only at the presence level and could not be verified for correctness. Future work should build ecosystem-specific ground truth pipelines (e.g., `cargo tree` for Rust, `go mod graph` for Go, `CocoaPods`/`Swift Package Manager` resolution for Swift) to confirm whether the patterns we observe at scale also hold for component and version accuracy in these ecosystems.

Tracing license metadata gaps to upstream registries. Our analysis in Section 5.4 attributed most missing license information to non-exact versioning or to package-manager types that the GitHub SBOM Tool does not resolve, but a substantial fraction of unlicensed dependencies in Java (42.2%) and PHP (49.1%) remained unaccounted for even though they carried exact versions and belonged to supported ecosystems. Future work could trace these specific cases directly to the upstream registries (Maven Central, Packagist) to determine whether the license metadata is missing at the source, inconsistently formatted, or simply not queried by the underlying SBOM generation tools, which would clarify whether the fix belongs to the SBOM tool or to the package registry itself.

Hybrid SBOM generation combining multiple tools. Our component-agreement analysis (Section 5.2) and accuracy evaluation show that no single tool dominates across all metadata dimensions: the GitHub SBOM Tool tends to surface more components and identifiers, while the Microsoft SBOM Tool more reliably resolves exact versions. This complementarity suggests an opportunity

to design and evaluate a hybrid SBOM generation approach that merges the outputs of multiple generators, for example, by using one tool's component list as the base and reconciling version and license fields from the most accurate source per ecosystem, and to measure whether such a merged SBOM improves downstream vulnerability and license-compliance outcomes beyond what any individual tool achieves.

Longitudinal analysis of SBOM quality. Our study captures a single snapshot of each repository. Building infrastructure that periodically captures GitHub-generated SBOMs over time would enable a longitudinal study of how SBOM completeness evolves as a project's dependency graph grows, as maintainers adopt stricter version pinning, or as GitHub updates its underlying SBOM generation mechanism. As discussed in Chapter 6 (Section 6.1), our sample was also not labeled by application domain or criticality; pairing a longitudinal design with such labels would help test whether SBOM maturity evolves differently across project domains, e.g., whether security-critical infrastructure projects converge on complete, well-versioned SBOMs faster than other projects.

Developer-facing study of SBOM usefulness. Finally, our evaluation is entirely tool-centric and does not capture how practitioners actually use SBOMs in vulnerability triage or license auditing workflows. A complementary study, e.g., a survey or interviews with developers and security teams who consume SBOMs, would help validate whether the completeness and consistency gaps identified in this thesis translate into real obstacles in practice, and would help prioritize which of the above directions matter most for practitioners.

Bibliography

[1] Ying Wang, Bihuan Chen, Kaifeng Huang, Bowen Shi, Congying Xu, Xin Peng, Yijian Wu, and Yang Liu. An empirical study of usages, updates and risks of third-party libraries in java projects. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 35–45, 2020.

[2] Piergiorgio Ladisa, Henrik Plate, Matias Martinez, and Olivier Barais. SoK: Taxonomy of Attacks on Open-Source Software Supply Chains . In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1509–1526, Los Alamitos, CA, USA, May 2023. IEEE Computer Society.

[3] William Enck and Laurie Williams. Top five challenges in software supply chain security: Observations from 30 industry and government organizations. *IEEE Security & Privacy*, 20(2):96–100, 2022.

[4] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. Backstabber's knife collection: A review of open source software supply chain attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 17th International Conference, DIMVA 2020, Lisbon, Portugal, June 24–26, 2020, Proceedings*, page 23–43, Berlin, Heidelberg, 2020. Springer-Verlag.

[5] Fabio Massacci and Ivan Pashchenko. Technical leverage in a software ecosystem: Development opportunities and security risks. In *Proceedings of the 43rd International Conference on Software Engineering*, ICSE '21, page 1386–1397. IEEE Press, 2021.

[6] Haya Samaana, Diego Elias Costa, Ahmad Abdellatif, and Emad Shihab. Opportunities and security risks of technical leverage: A replication study on the npm ecosystem. *Empirical Software Engineering*, 30(4), April 2025.

[7] Cybersecurity and Infrastructure Security Agency. Mitigating log4shell and other log4j-related vulnerabilities. Technical report, U.S. Department of Homeland Security, December 2021. Joint Cybersecurity Advisory AA21-356A.

[8] The Linux Foundation. What is an sbom? `https://www.linuxfoundation.org/blog/blog/what-is-an-sbom`, June 2021. Accessed: 2026-02-10.

[9] Boming Xia, Dawen Zhang, Yue Liu, Qinghua Lu, Zhenchang Xing, and Liming Zhu. Trust in software supply chains: Blockchain-enabled sbom and the aibom future. In *Proceedings of the 2024 ACM/IEEE 4th International Workshop on Engineering and Cybersecurity of Critical Systems (EnCyCriS) and 2024 IEEE/ACM Second International Workshop on Software Vulnerability*, EnCyCriS/SVM '24, page 12–19, New York, NY, USA, 2024. Association for Computing Machinery.

[10] Cybersecurity and Infrastructure Security Agency. SBOM FAQ 2024. Technical report, U.S. Department of Homeland Security, July 2024. Accessed: 2026-02-10.

[11] GitHub. What is an sbom (software bill of materials)? `https://github.com/resources/articles/what-is-an-sbom-software-bill-of-materials`, 2025. Accessed: 2026-02-10; Published: September 5, 2025.

[12] Executive Office of the President United States. Improving the nation's cybersecurity. Executive Order No. 14028, 86 Fed. Reg. 26633–26647, May 2021.

[13] National Telecommunications and Information Administration. The minimum elements for a software bill of materials (sbom). Technical report, U.S. Department of Commerce, July 2021.

[14] GitHub. Exporting a software bill of materials for your repository. `https://docs.github.com/en/code-security/supply-chain-security/understanding-your-software-supply-chain/`

exporting-a-software-bill-of-materials-for-your-repository, 2025. Accessed: 2025-07-03.

[15] Sabato Nocera, Simone Romano, M. D. Penta, R. Francese, and G. Scanniello. Software bill of materials adoption: A mining study from github. *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 39–49, 2023.

[16] Anesu Chaora, Nathan Ensmenger, and L Jean Camp. Discourse, challenges, and prospects around the adoption and dissemination of software bills of materials (sboms). In *2023 IEEE International Symposium on Technology and Society (ISTAS)*, 2023.

[17] National Telecommunications and Information Administration. Software bill of materials (sbom) overview. U.S. Department of Commerce, NTIA, August 2020. PDF available at https://www.ntia.gov/sites/default/files/publications/sbom_overview_20200818_0.pdf, accessed 2025-07-03.

[18] N. Zahan, Elizabeth Lin, Mahzabin Tamanna, W. Enck, Laurie A. Williams, D. Balzarotti, W. Enck, Thorsten Holz, and A. Stavrou. Software bills of materials are required. are we there yet? *IEEE Security & Privacy*, 21:82–88, 2023.

[19] The Linux Foundation. Spdx overview. https://spdx.dev/about/overview/, 2025. Accessed: 2025-07-03.

[20] NTIA Multistakeholder Process on Software Component Transparency Standards and Formats Working Group. Survey of existing sbom formats and standards. Technical report, National Telecommunications and Information Administration (NTIA), 2021.

[21] The Linux Foundation. Spdx 2.3 document overview. https://spdx.dev/learn/overview/spdx-2-3-document/, 2025. Accessed: 2025-07-08.

[22] GitHub. Rest api: Sboms. https://docs.github.com/en/rest/dependency-graph/sboms?apiVersion=2022-11-28, 2025. Accessed: 2025-07-10.

[23] National Institute of Standards and Technology. Executive order 14028, improving the nation's cybersecurity, 2022. Accessed: 2025-07-04.

[24] The White House. Executive order on improving the nation's cybersecurity, May 2021. Accessed: 2025-07-04.

[25] Framing Working Group NTIA Multistakeholder Process on Software Component Transparency. Framing software component transparency: Establishing a common software bill of material (sbom). Technical report, November 2019.

[26] Boming Xia, Tingting Bi, Zhenchang Xing, Qinghua Lu, and Liming Zhu. An empirical study on software bill of materials: Where we stand and the road ahead. *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2630–2642, 2023.

[27] Stephen Hendrick. The state of software bill of materials (sbom) and cybersecurity readiness. Research report, The Linux Foundation, 2022. Foreword by Jim Zemlin.

[28] Sheng Yu, Wei Song, Xunchao Hu, and Heng Yin. 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)*, pages 29–36, 2024.

[29] Chengjie Wang, Jingzheng Wu, Hao Lyu, Xiang Ling, Tianyue Luo, Yanjun Wu, and Chen Zhao. A large scale empirical analysis on the adherence gap between standards and tools in sbom. *ACM Trans. Softw. Eng. Methodol.*, January 2026.

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

[31] Sabato Nocera, Simone Romano, Massimiliano Di Penta, Rita Francese, and Giuseppe Scanniello. On the adoption of software bill of materials in open-source software projects. *Journal of Systems and Software*, 230:112540, 2025.

[32] Trevor Stalnaker, Nathan Wintersgill, Oscar Chaparro, M. D. Penta, Daniel M German, and D. Poshyvanyk. Boms away! inside the minds of stakeholders: A comprehensive study of bills of materials for software systems. *ArXiv*, abs/2309.12206, 2023.

[33] Gregorio Dalia, Corrado Aaron Visaggio, Andrea Di Sorbo, and Gerardo Canfora. Sbom ouverture: What we need and what we have. In *Proceedings of the 19th International Conference on Availability, Reliability and Security*, ARES '24, New York, NY, USA, 2024. Association for Computing Machinery.

[34] Tingting Bi, Boming Xia, Zhenchang Xing, Qinghua Lu, and Liming Zhu. On the way to sboms: Investigating design issues and solutions in practice. *ACM Trans. Softw. Eng. Methodol.*, 33(6), June 2024.

[35] Seth Carmody, Andrea Coravos, Ginny Fahs, Audra Hatch, J. Medina, Beau Woods, and J. Corman. Building resilient medical technology supply chains with a software bill of materials. *NPJ Digital Medicine*, 4, 2021.

[36] Aman Sharma, Martin Wittlinger, Benoit Baudry, and Martin Monperrus. Sbom.exe: Countering dynamic code injection based on software bill of materials in java, 2024.

[37] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. Germán, and Daniela Damian. The promises and perils of mining github. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR)*. ACM/IEEE, 2014.

[38] TIOBE Software. Tiobe programming community index. `https://www.tiobe.com/tiobe-index/`, 2025. Accessed: 2025-12-24.

[39] GitHub. Octoverse 2024: The most popular programming languages. `https://github.blog/news-insights/octoverse/octoverse-2024/`, 2024. Published: 2024-10-29; updated: 2025-10-28; Accessed: 2025-12-24.

[40] Stack Overflow. Stack overflow developer survey 2025: Technology. `https://survey.stackoverflow.co/2025/technology`, 2025. Accessed: 2025-12-24.

[41] Michail D. Papamichail, Themistoklis Diamantopoulos, Vasileios Matsoukas, Christos Athanasiadis, and Andreas L. Symeonidis. Towards extracting the role and behavior of contributors in open-source projects. In *Proceedings of the 14th International Conference on Software Technologies (ICSOFT 2019)*, pages 536–543, 2019.

[42] Abbas Javan Jafari, Diego Elias Costa, Rabe Abdalkareem, Emad Shihab, and Nikolaos Tsantalis. Dependency smells in javascript projects. *IEEE Transactions on Software Engineering*, 48(10):3790–3807, 2022.

[43] SEART Research Group. Seart github search (ghs). `https://seart-ghs.si.usi.ch/`, 2021. Accessed: 2025-12-24.

[44] Ozren Dabic, Emad Aghajani, and Gabriele Bavota. Sampling projects in github for msr studies. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 560–564, 2021.

[45] Alexandre Decan, Tom Mens, Pooya Rostami Mazrae, and Mehdi Golzadeh. On the use of github actions in software development repositories. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 235–245, 2022.

[46] Antonio Mastropaolo, Matteo Ciniselli, Luca Pascarella, Rosalia Tufano, Emad Aghajani, and Gabriele Bavota. Towards summarizing code snippets using pre-trained transformers. In *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*, ICPC '24, page 1–12, New York, NY, USA, 2024. Association for Computing Machinery.

[47] Joseph Romeo, Marco Raglianti, Csaba Nagy, and Michele Lanza. Uml is back. or is it? investigating the past, present, and future of uml in open source software. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 2342–2354, 2025.

[48] Hudson Borges, Andre Hora, and Marco Tulio Valente. Understanding the factors that impact the popularity of github repositories. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 334–344, 2016.

[49] Anchore. Syft: Sbom generation tool. `https://anchore.com/opensource/syft/`, 2025. Accessed: 2025-11-25.

[50] Aqua Security. Trivy: Open source vulnerability and sbom scanner. `https://trivy.dev/`, 2025. Accessed: 2025-11-25.

[51] Microsoft. Microsoft sbom tool. https://github.com/microsoft/sbom-tool, 2025. Accessed: 2025-11-25.

[52] OX Security. Top 5 sbom tools for securing the software supply chain. https://www.ox.security/blog/sbom-tools/, 2025. Accessed: 2026-01-21.

[53] Swaroop Sham. The top 11 open-source sbom tools. https://www.wiz.io/academy/application-security/top-open-source-sbom-tools, Dec 2025. Accessed 2026-01-21.

[54] Aqua Security. Trivy documentation: Repository scanning. https://trivy.dev/docs/latest/target/repository/, 2025. Accessed: 2025-11-25.

[55] Anchore. Syft: Supported sources. https://github.com/anchore/syft/wiki/supported-sources, 2025. Accessed: 2025-11-25.

[56] Microsoft. Sbom tool: Download and installation. https://github.com/microsoft/sbom-tool?tab=readme-ov-file#download-and-installation, 2025. Accessed: 2025-11-25.

[57] ClearlyDefined. ClearlyDefined: Helping foss projects be more clearly defined. https://clearlydefined.io/, 2025. Accessed: 2025.

[58] Weiwei Xu, Hao He, Kai Gao, and Minghui Zhou. Understanding and remediating open-source license incompatibilities in the pypi ecosystem. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 178–190, 2023.

[59] Amir M. Mir, Mehdi Keshani, and Sebastian Proksch. On the effect of transitivity and granularity on vulnerability propagation in the maven ecosystem. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 201–211, 2023.

[60] Alexandre Decan, Tom Mens, and Philippe Grosjean. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Softw. Engg.*, 24(1):381–416, February 2019.

[61] Jiaqi Wu, Lingfeng Bao, Xiaohu Yang, Xin Xia, and Xing Hu. A large-scale empirical study of open source license usage: Practices and challenges. In *Proceedings of the 21st International Conference on Mining Software Repositories*, MSR '24, page 595–606, New York, NY, USA, 2024. Association for Computing Machinery.

[62] Eduard Frankford, Tobias Antensteiner, Michael Vierhauser, Clemens Sauerwein, Vivien Wallner, Iris Groher, Reinhold Plösch, and Ruth Breu. A survey on feedback types in automated programming assessment systems. *ACM Trans. Comput. Educ.*, 26(1), December 2025.

[63] Jiaqi Liu, Qi Huang, Xin Xia, et al. An exploratory study on the introduction and removal of different types of technical debt in deep learning frameworks. *Empirical Software Engineering*, 26(1):16, 2021.

[64] William H. Kruskal and W. Allen Wallis. Use of ranks in one-criterion variance analysis. *Journal of the American Statistical Association*, 47(260):583–621, 1952.

[65] Olive Jean Dunn. Multiple comparisons among means. *Journal of the American Statistical Association*, 56(293):52–64, March 1961.

[66] OSV Project Maintainers. Osv.dev – open source vulnerabilities database. `https://osv.dev/`, 2025. Accessed: 2025-11-24.

[67] Python Software Foundation. Python package index (pypi). `https://pypi.org/`, 2026. Accessed: 2026-06-01.

[68] npm, Inc. npm. `https://www.npmjs.com/`, 2026. Accessed: 2026-06-01.

[69] Tidelift, Inc. Libraries.io. `https://libraries.io/`, 2026. Accessed: 2026-06-01.

[70] Federico Tomassetti and Marco Torchiano. An empirical assessment of polyglot-ism in GitHub. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering (EASE)*. ACM, 2014.

[71] Wen Li, Li Li, and Haipeng Cai. How are multilingual systems constructed: Characterizing language use and selection in open-source multilingual software. *ACM Transactions on Software Engineering and Methodology*, 33(3), 2023.