Master of Science in Software Engineering
January 2024

Exploring the Dynamics of Software Bill of Materials (SBOMs) and Security Integration in Open Source Projects

Anvesh Ambala

Faculty of Computing, Blekinge Institute of Technology, 371 79 Karlskrona, Sweden

This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Masters in Software Engineering The thesis is equivalent to 20 weeks of full-time studies.

The authors declare that they are the sole authors of this thesis and that they have not used any sources other than those listed in the bibliography and identified as references. They further declare that they have not submitted this thesis at any other institution to obtain a degree.

Contact Information:
Author(s):
Anvesh Ambala
E-mail: anam21@student.bth.se

University advisor:
Professor Davide Fucci
Department of Software Engineering

Faculty of Faculty
Blekinge Institute of Technology
SE–371 79 Karlskrona, Sweden

Internet : www.bth.se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57

Abstract

Background. The rapid expansion of open-source software has introduced significant security challenges, particularly concerning supply chain attacks. Software supply chain attacks, such as the NotPetya attack, have underscored the critical need for robust security measures. Managing dependencies and protecting against such attacks have become important, leading to the emergence of Software Bill of Materials (SBOMs) as a crucial tool. SBOMs offer a comprehensive inventory of software components, aiding in identifying vulnerabilities and ensuring software integrity.
Objectives. Investigate the information contained within SBOMs in Python and Go repositories on GitHub. Analyze the evolution of SBOM fields over time to understand how software dependencies change. Examine the impact of the US Executive Order of May 2021 on the quality of SBOMs across software projects. Conduct dynamic vulnerability scans in repositories with SBOMs, focusing on identifying types and trends of vulnerabilities.
Methods. The study employs archival research and quasi-experimentation, leveraging data from GitHub repositories. This approach facilitates a comprehensive analysis of SBOM contents, their evolution, and the impact of policy changes and security measures on software vulnerability trends.
Results. The study reveals that SBOMs are becoming more complex as projects grow, with Python projects generally having more components than Go projects. Both ecosystems saw reductions in vulnerabilities in later versions. The US Executive Order of 2021 positively impacted SBOM quality, with measures like structural elements and NTIA guidelines showing significant improvements post-intervention. Integrating security scans with SBOMs helped identify a wide range of vulnerabilities. Projects varied in critical vulnerabilities, highlighting the need for tailored security strategies. CVSS scores and CWE IDs provided insights into vulnerability severity and types.
Conclusions. The thesis highlights the crucial role of SBOMs in improving software security practices in open-source projects. It shows that policy interventions like the US Executive Order and security scans can significantly enhance SBOM quality, leading to better vulnerability management and detection strategies. The findings contribute to the development of robust dependency management and vulnerability detection methodologies in open-source software projects.

Keywords: Software Security, Supply chain attacks, Software Bill of Materials(SBOM), Open Source Projects, Vulnerability Management.

Acknowledgments

I extend my deepest gratitude to my supervisor, Davide Fucci, for his invaluable support throughout my thesis journey. His interactive approach and engaging discussions have greatly enriched my learning experience. The knowledge training sessions under his guidance were so good and played an important role in shaping my research. Davide's continuous encouragement, insightful suggestions, and constructive feedback have always been great. I am sincerely thankful for his support in every aspect of my work.

Last but certainly not least, I owe a heartfelt thanks to my family. To my mother, Ambala Sulochana, and father, Ambala Krishna, for their endless love and moral support. Their belief in me has been a constant source of strength. I also want to express my appreciation to my brother, Ambala Abhilash, for his encouragement and support. Thank you all for being my pillars of strength.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The domain of software development has been evolving continuously driven by more innovations and also increasing complexity of modern software systems. Open-source software has gained more popularity in recent years which gives many advantages including cost-effective software systems and allowing innovation. Open-source is a good strategy to make software development better, improving how good, reliable, and efficient software is made [1]. While the open-source approach enhances software development by making it more collaborative and efficient, it also introduces challenges, especially in terms of software security, specifically supply chain attacks. The extensive reliance on OSS introduces additional risks through external dependencies, necessitating comprehensive governance efforts across the entire software supply chain. OSS is a software that is licensed under an open-source license.

Software supply chain attacks, involving the introduction of vulnerabilities or malware in third-party components, during software development, pose significant risks when exploited by malicious actors [2]. To accelerate product development, companies often leverage open-source libraries, repositories of tools in various computer languages, such as npm [1] for node.js, Maven [2] for Java, or pypi [3] for Python. Recent instances highlight attempts to compromise these repositories, providing unauthorized access to a company's sensitive code and computer systems. In these attacks, cybercriminals create deceptive packages through package managers containing harmful code, including viruses or password stealers, and distribute them through website hosting software tools and third-party libraries. Failure by companies to thoroughly vet these tools can result in widespread impact, affecting millions of users [3]. For example, the UAParser.js npm package, a widely used tool for parsing user agent strings, was compromised in October 2021, leading to the theft of user credentials from applications relying on the package [4] [5]. Another notable incident involves the Log4Shell vulnerability discovered in the Log4j logging library in December 2021, enabling malicious actors to seize control of vulnerable systems through specially crafted log messages. These attacks underscore the risk of hackers inserting malicious code into software, especially as numerous software projects heavily depend on diverse open-source packages that may contain undisclosed vulnerabilities [6] [7].

When it comes to software engineering, managing dependencies involves making

[1] `https://docs.npmjs.com/`
[2] `https://maven.apache.org/`
[3] `https://pypi.org/`

decisions about the parts of the software that are not developed by the company and come from external sources. In today's world, these dependencies can be significant, such as complete operating systems like Docker or even entire infrastructures known as "Infrastructure as Code" or IaC. However, this creates new challenges when it comes to protecting against supply-chain attacks. Currently, the most prevalent approach for addressing supply-chain attacks is to create a Software Bill of Material (SBOM) - it is like a catalog of all the parts, how they are arranged, and their characteristics. We prepare it so that it can be used later to fix and maintain products that have already been deployed. [8]. While the SBOM is crucial for identifying and remediating vulnerabilities in software dependencies, it is essential to note that it does not eliminate the need for the thorough evaluation of software components. Instead, it streamlines the evaluation process by providing a structured and standardized format for sharing results. This structured format enhances collaboration, allowing stakeholders to easily interpret and act upon evaluation findings. Prepared with meticulous detail, an SBOM acts as a valuable tool for fixing and maintaining products post-deployment, functioning akin to an inventory or list of ingredients that are crucial in the software development process [9]. However, all SBOMs share the common goal of providing a comprehensive inventory of the software components used in a software project. This information is essential for vulnerability management, as it enables organizations to identify and remediate vulnerabilities in their software dependencies.

The importance of SBOMs was recently recognized by the U.S. government, which issued Executive Order 14028 on Improving the Nation's Cybersecurity. This order directs federal agencies to adopt SBOMs for all software they develop or procure, and it encourages private-sector adoption. As a result, the use of SBOMs is on the rise, and organizations are increasingly recognizing their value in improving software security [10].

Vulnerability management is a significant reason why SBOMs are essential. Currently, identifying if a specific software component has an issue and whether that issue could affect other connected parts can be time-consuming and expensive. However, SBOM data simplifies the process by enabling suppliers, users, and other individuals responsible for software security to promptly and accurately evaluate the risks posed by these vulnerable components that may be hidden in the supply chain. It's similar to the concept that you can't protect something unless you know what it is [9]. The purpose of this research is to investigate the Software Bill of Materials (SBOMs) in the context of open-source software systems, with a specific focus on the Python and Go ecosystems hosted on GitHub. SBOMs are essential documents that provide information on the components and dependencies of software projects. This research aims to gain a comprehensive understanding of SBOMs by exploring their contents, evolution, impact of US Executive order and practical implications in enhancing software security.

This research will address key questions such as what information is included in SBOMs and associated vulnerability scans, how they evolve, and the significance of integrating security scans with SBOMs. Also, understanding the impact of US

Executive Order of May 2021 on the quality of SBOM.

1.1 Aims and Objectives

This thesis aims to explore and understand how Software Bill of Materials (SBOMs) is managed and how security measures are integrated in open-source projects, particularly in the Python and Go ecosystems on GitHub. This includes investigating the impact of policy intervention, like the US Executive Order, on SBOM quality across software projects. The research intends to provide insights for enhancing security practices in open-source development. The objectives of the thesis include:

- Investigate the information contained within Software Bill of Materials (SBOMs) in Python and Go repositories on GitHub.
- Analyze the evolution of SBOM fields over time to understand how software dependencies change.
- Analyze how the US Executive Order of May 2021 impacted the quality of SBOMs across software projects.
- Conduct vulnerability scans in repositories with SBOMs, focusing on identifying types and trends of vulnerabilities.

1.2 Research Questions

To address the objectives of this thesis work the following research questions have been formulated:

RQ1: What information is contained in SBOMs and associated vulnerability scans?

RQ1.1: How do SBOMs and associated vulnerability scans evolve?

Justification: This research question addresses the fundamental need to understand the information contained in SBOMs and vulnerability scans, which are essential artifacts for managing software supply chain security. By examining the nature of this information and its evolution over time, we can inform the development of effective strategies for dependency management and vulnerability detection.

RQ2: What is the impact of the US Executive Order of May 2021 mandate on the use of SBOMs on the improvement of quality of SBOMs across software projects?

Justification: This question examines if the U.S. Executive Order on SBOMs improves their quality in organizations. It's key to understanding how government actions can better software security. The findings from this study can help in making better policies and strategies for managing software vulnerabilities, similar to how

RQ1 explores the content and evolution of SBOMs and vulnerability scans for effective software management.

RQ3: How does incorporating results from security scan tools as part of SBOMs impact the identification of vulnerabilities across diverse open-source software projects?

Justification: This research question looks at how putting security scans together with SBOMs can help find and fix security problems in open-source software projects. By trying out this combined approach and seeing how well it works, we can give helpful advice to organizations that use open-source software to make their overall security better.

Chapter 2

Background

This section contains key terminologies and concepts that are part of this research.

2.1 Supply chain attack

Supply chain attacks target less secure elements in the software supply chain to breach an organization's systems. These attacks can occur in any industry, from finance to energy to government, and can target both software and hardware. Cybercriminals often tamper with the manufacturing or distribution of a product by embedding malware or hardware-based spying components [11]. The attack can then propagate from the directly impacted firms to their suppliers and customers, highlighting the importance of maintaining visibility and awareness within these complex supply chains. Supply chain attacks are becoming increasingly prevalent, posing a significant threat to business relationships with partners and suppliers. Their undetectable nature makes them difficult to detect, and just because a software product was once validated doesn't guarantee its continued security [12].

2.2 SBOM

Definition of SBOM: A Software Bill of Materials (SBOM) is a detailed and organized list that includes all the components, dependencies, and characteristics of a software product. It presents a standardized and transparent view of the software's essential components, such as open-source elements, third-party libraries, and proprietary code. This information can include the name and version of the component, the vendor of the component, and the license under which the component is distributed [13].
Formats in SBOM: SPDX and CyclonDX and the two main open-source software formats that are used to describe SBOM. CycloneDX was created in 2016 specifically for SBOMs while SPDX predates it and has broader reuse in areas like open source licenses. Both formats describe components, dependencies, vulnerabilities, licenses, etc. in a machine-readable format. CycloneDX focuses more on dependencies while SPDX covers a wider set of metadata. The adoption of CycloneDX and SPDX formats has increased in recent years among open-source projects and companies like Red Hat, GitHub, Intel, etc. This shows a growing recognition of the importance of SBOMs [14].

The Role of SBOM in Securing the Software Supply Chain:
SBOMs can play a critical role in securing the software supply chain in a number of ways [13].

- Identifying vulnerabilities: SBOMs can be used to identify vulnerabilities in the software supply chain. This information can be used to prioritize remediation efforts and to prevent vulnerabilities from being exploited.
- Tracking dependencies: SBOMs can be used to track dependencies between software components. This information can be used to identify potential vulnerabilities that could be introduced by updates to dependencies.
- Improving communication: SBOMs can improve communication between developers and security teams. This information can be used to identify and address security concerns early in the development process.

Benefits of SBOM There are a number of benefits to using SBOMs [13].

- Increased visibility: SBOMs provide increased visibility into the software supply chain. This can help to identify and address potential security risks.
- Reduced risk: SBOMs can help to reduce the risk of supply chain attacks. This is because they provide information that can be used to identify and mitigate vulnerabilities.
- Improved compliance: SBOMs can help to improve compliance with security requirements. This is because they provide information that can be used to demonstrate that software is secure.

2.3 Open-source software

It is a software that is licensed under an open-source license. The concept originated in the 1970s, and OSS has since revolutionized software development by enabling users to run, study, modify, and distribute the software freely [15].

2.3.1 Github

GitHub is a cloud-based platform for software development and version control, enabling developers to store and manage their code [16]. Its user-friendly interface makes it accessible to individuals and teams, facilitating collaboration and version control using Git [17].

2.3.2 Python and Go ecosystems

Both Python and Go ecosystems are widely used and vibrant in software development. The Python ecosystem is a collection of tools, libraries, and frameworks that are used for scientific computing. The Python ecosystem is built around the Python programming language, which is a general-purpose language that is well-suited for

scientific computing [18] The GO ecosystem is built around the Go programming language, which is a compiled, statically typed language that is well-suited for developing microservices-based applications [19].

2.4 Vulnerability scan

Vulnerability scanning constitutes an essential component of cybersecurity, enabling organizations to identify and remediate potential weaknesses in their systems before attackers can exploit them. There are several tools available for scanning vulnerabilities, such as Synk, OSV scanner, Grype, Nikto, and NMap. Reports generated from vulnerability scans provide a lot of valuable information that can be used to remediate vulnerabilities. This includes the identification of vulnerabilities, guidance on how to fix them, prioritization of the vulnerabilities based on their severity, and compliance reporting. Regular vulnerability scanning and remediation help prevent data breaches and maintain the security of an organization's digital assets. These automated processes scan software for weaknesses, flaws, or vulnerabilities, proactively addressing potential security risks [20].

2.4.1 OSV scanner

The OSV-Scanner is a vulnerability scanner written in Go that provides an officially supported frontend to the OSV database. Its purpose is to discover existing vulnerabilities affecting a project's dependencies. The OSV database serves as a distributed vulnerability database for open-source ecosystems, and the OSV Scanner links a project's dependency list with the vulnerabilities that affect them. It utilizes the OSV schema, which provides a human and machine-readable data format for describing vulnerabilities in a way that precisely maps to open-source package versions or commit hashes [21].
To access the OSV website you can use the following URL: https://OSV.dev/

2.4.2 CVSS Score

The Common Vulnerability Scoring System (CVSS) is a standardized method for assessing the severity of software vulnerabilities. CVSS is used by security professionals, software vendors, and government agencies to prioritize vulnerability remediation efforts. CVSS version 2 was released in 2006 and is the current version of the standard. CVSS version 2 is a more comprehensive and flexible system than its predecessor, CVSS version 1 [22].

CVSS Scoring Components CVSS version 2 scores vulnerabilities based on four components:

- **Base Score:**The base score is a numeric value that represents the intrinsic characteristics of a vulnerability.
- **Temporal Score:** The temporal score is a numeric value that represents the current risk associated with a vulnerability.

- **Environmental Score:** The environmental score is a numeric value that represents the specific context of a vulnerability.
- **Overall Score:** The overall score is the sum of the base, temporal, and environmental scores.

2.4.3 Common Weakness Enumeration (CWE)

The Common Weakness Enumeration (CWE) is a list of software weaknesses that can be exploited by attackers to gain unauthorized access to systems or data. CWE is a widely used standard for identifying, classifying, and categorizing software weaknesses [23].
To access the CWE-ID website you can use the following URL: https://cwe.mitre.org/index.html

Purpose of CWE: CWE is a tool that can be used to identify vulnerabilities in software systems. By using this tool, developers can prioritize their remediation efforts more effectively. This is because CWE provides a common language for communicating about software weaknesses, which can help to improve communication between developers, security professionals, and vendors. By using CWE, developers can identify the most critical vulnerabilities and ensure that they are addressed first, thereby reducing the risk of security breaches.

Structure of CWE: CWE is a hierarchical taxonomy of weaknesses. The top level of the taxonomy is divided into the following categories:

- Input Validation: Weaknesses in input validation can allow attackers to inject malicious code into systems.
- Data Handling: Weaknesses in data handling can allow attackers to access or modify sensitive data.
- Resource Management: Weaknesses in resource management can allow attackers to exhaust system resources or gain unauthorized access to systems.
- Security Features: Weaknesses in security features can allow attackers to bypass security controls.
- Architectural Issues: Weaknesses in architectural issues can make systems more vulnerable to attack.

Benefits of Using CWE: There are several benefits to using CWE, including:

- Standardized language: CWE provides a standardized language for communicating about software weaknesses. This can help to improve communication between developers, security professionals, and vendors.
- Improved vulnerability management: CWE can be used to improve vulnerability management processes. This can help to ensure that vulnerabilities are identified, classified, and prioritized in a consistent manner.
- Reduced risk: CWE can help to reduce the risk of software vulnerabilities being exploited. This can help to protect organizations from cyberattacks.

2.5 US Executive Order May 12, 2021

The Executive Order on Improving the Nation's Cybersecurity, which was issued by President Biden on May 12, 2021, is a comprehensive framework for addressing the cybersecurity challenges facing the United States. The order identifies four key areas of focus [24]:

1. **Improving software supply chain security:** The order directs the federal government to take steps to secure the software supply chain, including developing and adopting standards for SBOMs and requiring federal agencies to use SBOMs when procuring software.

2. **Enhancing threat intelligence sharing:** The order directs the federal government to enhance its sharing of threat intelligence with the private sector, and to establish a national cybersecurity risk management framework.

3. **Strengthening cybersecurity research and development:** The order directs the federal government to invest in cybersecurity research and development, and to promote international collaboration on cybersecurity.

4. **Modernizing cybersecurity defenses:** The order directs the federal government to modernize its cybersecurity defenses, including implementing new technologies and tools, and improving cybersecurity training for federal employees.

The Executive Order on Improving the Nation's Cybersecurity is a significant step forward in the Biden administration's efforts to protect the United States from cyberattacks. The order's focus on SBOMs is particularly important, as SBOMs can provide valuable information about the software that is used in critical systems, and can help to identify and mitigate vulnerabilities. Here are some of the specific actions that the Executive Order directs agencies to take [24]:

- **Develop and adopt standards for SBOMs:** The order directs the National Institute of Standards and Technology (NIST) to develop and adopt standards for SBOMs. These standards will help to ensure that SBOMs are interoperable and that they can be used to effectively identify and mitigate vulnerabilities.

- **Require federal agencies to use SBOMs when procuring software:** The order directs federal agencies to require that vendors of software provide SBOMs with their products. This will give federal agencies the information they need to make informed decisions about the software they purchase and to mitigate the risk of cyberattacks.

- **Encourage the private sector to use SBOMs:** The order encourages the private sector to use SBOMs to improve the security of their software. The order also directs the Department of Commerce to work with industry groups to promote the adoption of SBOMs.

To look into the US Executive order of May 2021 you can use the following URL: https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity/

Chapter 3

Related Work

A detailed analysis of relevant literature shows that SBOMs play a critical role in improving software supply chain security. Xia et al. [39] em- phasize the importance of SBOMs as a tool to increase transparency in the complex systems of software supply chain in light of the growing risk of software supply chain attacks. Their research highlights a significant gap in knowledge among practitioners about the challenges in adopting SBOMs. Moreover, Xia et al. [39] offer a goal model designed specifically for practitioners, which presents a systematic roadmap for the adoption of SBOMs, making a valuable contribution to the field.

Turning their attention to adoption patterns, Nocera et al. [25] examine the state of SBOM utilization in open-source software projects at the moment. According to their research, there is a slight but growing trend in the usage of SBOMs, which suggests that developers are becoming more conscious of the value of risk reduction and visibility in the software supply chain. Building on these discoveries, Stalnaker et al. [26] conducted a thorough analysis of the difficulties experienced by parties involved in the creation and use of SBOM. The authors acknowledge these difficulties and add to the current discussion on strengthening software supply chain security by outlining workable solutions and future research directions. By addressing practitioner attitudes, adoption trends, and problems, these publications collectively offer a detailed understanding of SBOMs and lay a crucial foundation for the advancement of security practices across the software supply chain.

Bi et al., [27] and Buttner et al. [28] discuss the importance of SBOMs in strengthening supply chain security. Bi et al., [27] analyzed 4,786 GitHub talks from 510 projects and uncovered on important topics, problems, and solutions connected to SBOM practices. The paper explores commonly used frameworks and methods for creating SBOMs, including a comprehensive analysis of their benefits and limitations. The findings highlight the importance of SBOMs in upholding strong software development practices and the need for broad integration to improve supply chain security. Buttner et al. [28] highlight the useful advantages of SBOM implementation, explaining its ability to reduce risks related to unknown software and improve understanding of the makeup of critical infrastructure. The SBOM enables teams working on cybersecurity, acquisition, and system engineering to automate risk assessments, promoting informed decision-making. The research emphasizes the critical role that SBOMs play in supply chain security and offers useful data for further research and development in this important area. Collectively, they promote the broad implemen-

tation and integration of SBOMs to strengthen software transparency, cybersecurity protocols, and general software component reliability.

Chaora et al. [29] explored the integration of software with societal infrastructure and emphasized the need for strong software supply chain security. The paper highlights the widespread use of software in various sectors, including industrial, manufacturing, and municipal technologies, that necessitates a dependable and secure software supply chain. The authors discussed various initiatives to enhance risk management in software supply chains, such as SBOMs. However, the paper also highlights some challenges in SBOM distribution, including a lack of support and enthusiasm. The paper also outlines the benefits of SBOM adoption, including the mitigation of risks associated with unknown software and an increase in the comprehension of critical infrastructure components. SBOMs enable teams involved in cybersecurity, acquisition, and system engineering to automate risk assessments, thereby fostering informed decision-making. The study emphasizes the critical role of SBOMs in supply chain security and provides valuable insights for future research and development in this domain. The authors advocate for the widespread implementation and integration of SBOMs to strengthen cybersecurity protocols and enhance software transparency and reliability of software components.

Caven et al. [30] and Ding et al. [8] together add to the discussion on SBOMs by looking at a variety of topics, from legislative actions to real-world applications. The regulatory environment in the US as described by Presidential Executive Order 14028 is the subject of Caven et al.'s [30] analysis, which highlights the necessity of both SBOMs and an extensive cybersecurity labeling program. Despite major modifications to these programs, the authors highlight the continued importance of SBOMs and labels, which are reinforced in the 2023 National Cybersecurity Strategy. The study emphasizes that while SBOMs and labels can work together to improve product security over the course of a product's lifecycle, their effectiveness is dependent on both application and dependability.

Ding et al. [8] researched how SBOMs are implemented in the real world using enterprise big data. They found that system integration can speed up the SBOM generation process by using the Engineering Bill of Materials (EBOM) for a particular product type. The researchers offer a comprehensive method for creating SBOMs by combining order data from Enterprise Resource Planning (ERP) and actual installation data from Manufacturing Execution Systems (MES). The research shows that this technique was able to significantly reduce the production cycle for SBOM generation from 10 to 20 days down to just 3 to 5 hours.

Ding et al. [8] offer valuable insights into the benefits and increased productivity that can be achieved through the integration of SBOMs into organizational operations. Meanwhile, Caven et al. [30] focused on the wider regulatory environment and emphasized the urgent need for SBOMs. The combined effects of practical implementation and regulatory action underscore the numerous advantages of SBOMs in enhancing transparency and cybersecurity within the supply chain.

Faruk et al. [2] go deeper into the nuances of software supply chain threats, building on the fundamental insights offered by Chaora et al. [29], who highlighted the crucial importance of SBOMs in increasing software supply chain security. While Faruk et al. [2] particularly address the downstream hazards associated with exploiting vulnerabilities during software manufacturing, Chaora et al. [29] underlined the rising integration of software into societal infrastructure and the significance of SBOMs. In line with the main subject of protecting the software supply chain, the study by Faruk et al. [2] adds to the story by highlighting the necessity of strong security measures within an organization's infrastructure and development tools.

In a comparable context, Fu et al. [3] investigate vulnerabilities in third-party package repositories further, reiterating worries about software supply chain attacks expressed by Faruk et al. [2]. Fu et al.'s study [3] focuses on the introduction of malicious code into legitimate packages found in repositories such as NPM, PyPI, or RubyGems, highlighting a particular way that software supply chain attacks can appear. Collectively, these research works provide a coherent story that advances from the fundamental significance of SBOMs and the wider consequences of software supply chain intrusions to particular cases of vulnerability in third-party repositories. This field offers a thorough comprehension of the risks and difficulties that are common in the ecosystem of the software supply chain.

Plate et al. [31] complement the broader discussions on software security and supply chain flexibility presented in previous studies. Building on the foundational aspects explored by Buttner et al. [28] regarding the adoption of an SBOM for transparency and Faruk et al. [2] addressing software supply chain assaults, Plate et al. [31] explore into the specific realm of open-source software (OSS) vulnerabilities. Their focus on assessing the impact of vulnerabilities through code modifications aligns with the overarching theme of understanding and mitigating risks within the software supply chain. By bridging the gap between the strategic considerations of SBOM adoption and the operational challenges posed by OSS vulnerabilities, this research field provides a comprehensive exploration of diverse dimensions crucial to enhancing software security in current development practices.

Expanding upon the examination of software supply chain obstacles, Gokkaya et al. [32] further develop the story by focusing on the increased dangers connected to software items being used as cyberattack vectors inside the software supply chain. Thorough risk assessment is crucial in the face of evolving attack vectors, building on previous discussions of SBOMs, regulatory actions, and vulnerability detection. Going beyond the acceptance and problems of regulatory frameworks, vulnerability mitigation measures, and software business models, Gokkaya et al.'s work adds a critical layer of analysis by concentrating on the type and traits of software supply chain attacks. When taken as a whole, these observations help to provide a more comprehensive understanding of the complex field of software supply chain security, opening the door to stronger and more reliable defenses for the software supply chain.

By presenting an automated method for vulnerability discovery using machine learning, the research by Harer et al. [33] offers a fresh viewpoint on enhancing software

security. This study adds to the larger conversation started by earlier research that examined problems related to software supply chain attacks, risk assessment techniques, and the use of SBOMs. While previous research has focused on the necessity of risk mitigation strategies such as frameworks like SBOMs and increased transparency in the software supply chain, Harer et al.'s study tackles the technical side of proactively finding vulnerabilities. The integration of machine learning in vulnerability detection aligns to enhance software security measures and provide a data-driven approach to strengthen software system flexibility. When taken as a whole, these studies add to a thorough grasp of the difficulties and developments in the field of software security, highlighting a variety of aspects such as automated detection, risk management, and transparency.

The deployment of SBOMs and their potential to improve software supply chain security is discussed in Eggers et al.'s study [34] in the context of the nuclear industry. This study reinforces previous research that highlights the importance of SBOMs in enhancing risk management and transparency in the software supply chain. The authors especially focused on the challenges faced by large industrial facilities, such as nuclear power plants, in implementing SBOMs due to the industry's conservative nature. This finding aligns with earlier research on the slow acceptance of SBOMs and the need for feasible implementation plans. Eggers et al. suggest a "crawl, walk, run" approach to gradually incorporate SBOMs into current procedures, which provides organizations with a clear roadmap to follow. This study contributes to the ongoing discussion on improving supply chain resilience and cybersecurity in the wider context of software security by proposing ideas that could be applied across a range of industrial sectors.

The study conducted by Zahan et al [35] explores the correlation between the number of vulnerabilities in PyPI and npm packages and software security practices. The study was driven by the need for more secure software, especially in the context of the U.S. Executive Order 14028. The OpenSSF Scorecard project was used to automatically measure the adoption of software security techniques. The study developed five supervised machine learning models, which demonstrated that vulnerability counts were significantly affected by four security practices: maintenance, code review, branch protection, and security policy. However, the models showed poor predictability when it came to anticipating vulnerability counts, with R*2 ranging from 9 to 12. Unexpectedly, the study found that when aggregate security ratings increased, reported vulnerability counts also increased, indicating the presence of other factors. The report highlights issues such as the lack of vulnerability data, the need for better detection scripts for security practices, and the time constraints associated with implementing security procedures. To provide more practical advice on security procedures, the authors recommend improving vulnerability count and security score data.

The topic of Software Bill of Materials (SBOM) is relatively new in the research domain, and many research attempts are underway to explore this topic. The existing body of literature on SBOM and software supply chain has provided valuable insights. Some research gaps have been identified, and the things that will be addressed in this thesis work are listed below:

- Limited understanding of SBOM evolution in GitHub Repositories: While existing studies acknowledge the significance of SBOM, there is a gap in understanding how SBOM evolves in projects. This will be addressed in this thesis work by looking at how SBOM fields evolve within the projects particularly concerning within the Python and Go ecosystems.

- Inadequate Assessment of Policy Impact on SBOM: While the importance of such policies like US Executive Order 2021 has been noticed but what impact did it play and how this policy has affected or increased the quality of the SBOM file has been not done, this thesis attempts to evaluate the impact that the order has created on quality of SBOM by comparing the quality of SBOM files before and after the executive order.

- Limited exploration of the combined impact of SBOM and security scan: Previous studies have shown that it's important to combine security scans with SBOMs to identify vulnerabilities across projects. However, there's still a gap in understanding how this integration impacts vulnerability identification. This study gives practical tips on integrating security scans with SBOMs. It explores the scanning tools available in the market and provides insights into what information from the scanning results can help identify vulnerabilities. The study also offers suggestions on how to improve the vulnerability identification process by integrating security scans with SBOMs.

Chapter 4

Method

4.1 Research method Selection

For this thesis work, two research methods were chosen: archival study and Quasi-experiment. Archival study is used to answer RQ1 and RQ3, while Quasi-Experiment is used to answer RQ2.

Archival study: Archival research is a type of research that involves searching for and extracting information and evidence from original archives, such as manuscripts, documents, records, objects, sound and audiovisual materials, or other materials. Archival research is typically conducted in archives, Special Collections libraries, or other repositories [36].

Justification: An archival study is well-suited to addressing RQ1 for several reasons:

Existing Data Availability: SBOMs and associated vulnerability scans are readily available from various sources, including open-source repositories, public datasets, and organizations that utilize these tools for software supply chain security management. This existing data allows for a comprehensive exploration of the information contained in SBOMs and vulnerability scans.

Longitudinal Analysis: SBOMs and vulnerability scans are typically generated over time, providing a source of data for analyzing their evolution. An archival study enables the tracking of changes in SBOM content, vulnerability patterns, and dependencies across different releases of software projects.

Comparative Analysis: By collecting SBOMs and vulnerability scans from diverse software projects and ecosystems, an archival study can facilitate a comparative analysis of their information content and evolution. This comparison can reveal differences and similarities in the way SBOMs and vulnerability scans are used and maintained across different software development practices.

In contrast, other research methods, such as surveys, are less suitable for addressing RQ 1. Surveys would require a significant effort to recruit and engage a large enough sample of software developers and security professionals to provide representative insights into the information content and evolution of SBOMs and vulnerability scans.

Other research methods like case studies are less suitable for answering rq3, while case studies can give us valuable information about specific examples of integrating security scans with SBOMs, they are not the best way to prove that this integration

approach works. This is because case studies are more like observing what happens in real-world situations, rather than setting up a controlled experiment.

In the context of RQ3, a case study would involve examining a few specific software projects that have integrated security scans with SBOMs and describing their vulnerability identification practices. While this could provide some interesting qualitative data, it would not allow us to definitively say whether the integration approach itself is the reason for any observed improvements or changes in vulnerability detection.

Quasi Experiment: A quasi-experiment is a study designed to assess the causal impact of an intervention on a target population without employing random assignment [37]. While bearing similarities to traditional experimental designs or randomized controlled trials, quasi-experimental research differs by not incorporating random assignment to treatment or control groups. In quasi-experimental designs, researchers usually have some degree of control over treatment assignment but employ criteria other than random allocation. [38].

Justification: A quasi-experimental design is a valuable method to study the impact of an intervention in situations where random assignment is not feasible. This method is used to address RQ2, which seeks to understand the impact of the US Executive Order of May 2021 on the use of Software Bill of Materials (SBOMs) across various software projects. The challenge lies in the real-world context of diverse software projects affected by a government order, making a full experimental setup with random assignments impractical.

4.2 Research Design

This section provides an overview of the direction employed in this thesis. The research design is depicted in Figure 4.1

As shown in the research design figure, the repository search, SBOM collection, and SBOM content analysis phases are part of the archival study. The remaining phases, vulnerability scanning tool selection, vulnerability scanning, evolution analysis, vulnerability result analysis, Python vs. Go ecosystem comparison, and SBOM analysis by criteria, are involved in the quasi-experiment.

4.3 Archival Study

4.3.1 Data Collection:

This research aims to collect SBOMs and vulnerability scan data from various open-source software projects in the Python and Go ecosystems. The selection of repositories will be based on their relevance, usage, and the availability of SBOM data. To evaluate the quality of SBOMs pre and post the US Executive Order, data will be collected from various projects existing both before and after the order. The

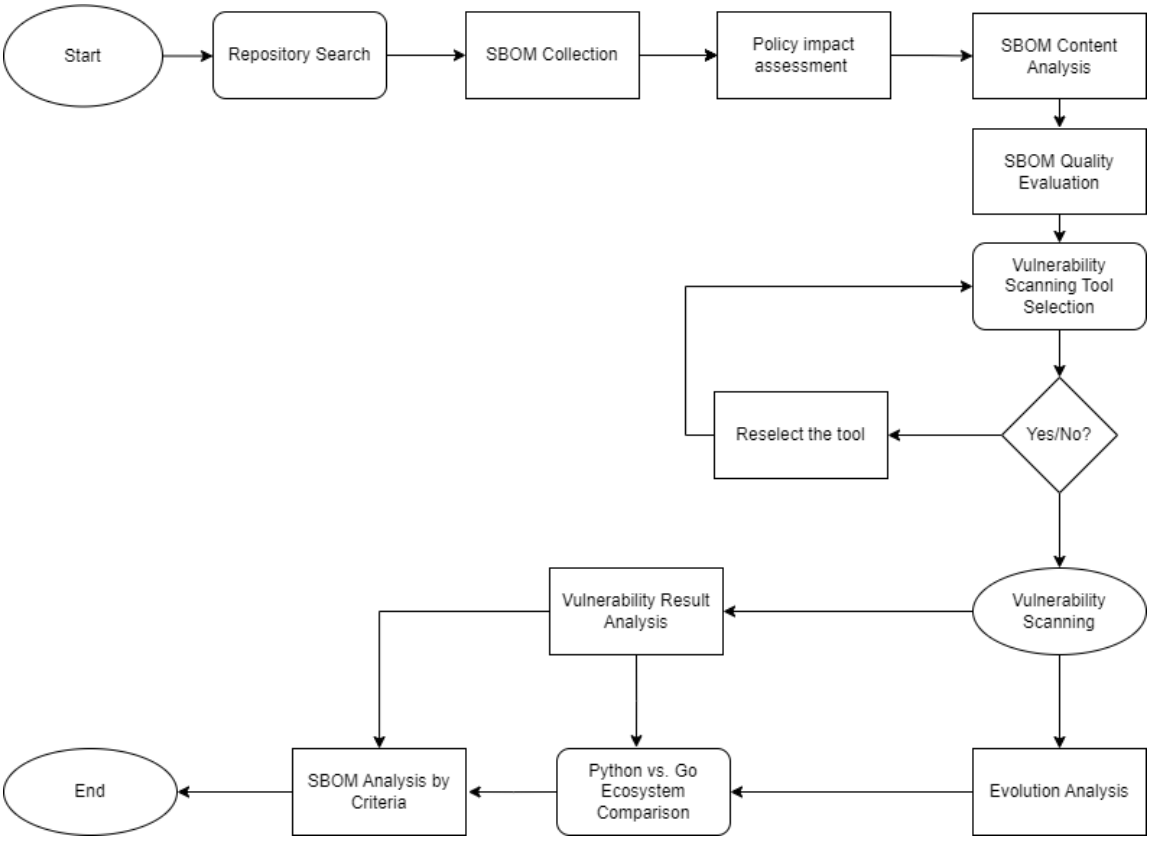


Figure 4.1: Research Design

collection methods will ensure that a comprehensive dataset is obtained for robust analysis in subsequent phases.

Repository search: To identify repositories containing SBOM files, a Python script was developed that utilized the GitHub API. The script first set up authentication using a GitHub personal access token, ensuring authorized access to the API. Next, it defined a list of SPDX/cycloneDx file extensions as search criteria. The search query was constructed to include repositories containing files with any of the specified extensions. The script then formulated the search URL and made an API request using the requests library. Upon receiving a successful response, it extracted the repository's full name from each search result and appended it to a list. If the API request failed, the script displayed an error message. Finally, it iterated through the list of repositories and printed each repository's full name to the console.

The query used to search for SBOMs in SPDX and CycloneDX formats within the Python and Go ecosystems involved searching GitHub repositories using the following query /search/repositories?q=language:<language>&filename:spdx OR filename:scyclonedx&per-page=100&page=<page>.

The repository search process was further enhanced by utilizing Sourcegraph, a code intelligence platform that provides a comprehensive search capability across various programming languages. Unlike the GitHub API approach, which was limited to searching for Python repositories with SPDX/CycloneDx extensions, Sourcegraph

enabled a broader search across all repositories, regardless of the programming language. This enhanced search strategy significantly expanded the pool of repositories containing SBOM files, providing a more comprehensive dataset for further analysis.

This thesis focuses on the analysis of SBOM files to assess the security posture of software projects within the Python and Go ecosystems. These two ecosystems were chosen due to their widespread adoption and popularity in modern software development. Python is a general-purpose programming language known for its ease of use and versatility, making it a popular choice for web development, data science, and machine learning applications. Go, on the other hand, is a relatively new programming language that has gained significant traction in recent years due to its emphasis on performance, concurrency, and reliability, making it well-suited for systems programming and microservices architecture.

The repository search process was further enriched by utilizing a dataset compiled from a bachelor thesis at the university, which provided a valuable list of SBOM-containing repositories. This external resource played a crucial role in complementing the repositories identified through GitHub API and Sourcegraph searches, ensuring a comprehensive dataset for analysis. Sincere gratitude is extended to the authors of the bachelor thesis for their valuable contribution to this research. Their efforts in compiling the dataset enabled a broader repository search, laying a solid foundation for analyzing SBOM usage and vulnerabilities in the Python and Go ecosystems

SBOM collection: Following the repository search, the next phase involved collecting the SBOM files associated with the identified repositories. For a total of 8 projects A.1 in Go and 8 projects in Python, SBOM file collection was carried out through a manual approach. This involved:

- **Repository Identification:** The repository names were identified from the comprehensive dataset compiled during the repository search phase.
- The SBOM collection process faced two challenges: managing multiple SBOM files within repositories and gathering SBOM files across multiple releases. To effectively address these complexities, a combined approach was employed, utilizing both manual and automated strategies.
- **Managing Multiple SBOM Files in Repositories:** When multiple SBOM files were found within a repository, careful attention was paid to identify and collect the relevant files corresponding to the desired software versions. This involved:
  Version Identification: Examining file names to extract version information and accurately associate SBOM files with specific software versions.

  Manual Selection and Download: Manually selecting and downloading the SBOM files corresponding to the desired software versions, ensuring that only relevant files were collected for further analysis.

  By employing these strategies, the collection process effectively addressed the

presence of multiple SBOM files within repositories and ensured the accurate identification of relevant files for further analysis.

- **Gathering SBOM Files Across Multiple Releases**

  A systematic approach was needed to collect SBOM files across multiple releases. To simplify the process, a Python script was created that automates the identification and downloading of SBOM files. This script uses the requests library to make API calls to the GitHub API. It navigates through releases efficiently, identifies relevant SBOM files, and downloads them to a specified directory.

  The script's functionality included:

  - Retrieving Release Information: Making API calls to the GitHub API to retrieve information about releases for the specified project.
  - Parsing JSON Response: Extracting relevant SBOM file URLs from the JSON response.
  - Downloading SBOM Files: Issuing API calls to download identified SBOM files and saving them to the designated directory.
  - Error Handling: Implementing error handling mechanisms to capture and report any download failures, ensuring the integrity of the collected SBOM files.

  By leveraging the Python script, the collection of SBOM files across multiple releases became more efficient and consistent, reducing manual effort and ensuring that all relevant SBOM files were collected for further analysis.

By carefully handling both scenarios – multiple SBOM files within repositories and SBOM files across multiple releases – the SBOM collection process successfully gathered a comprehensive dataset of SBOM files. An example of how an SBOM file will look like is shown here A.1

4.4 Quasi Experiment

In this study, the quasi-experimental research method is employed to understand the impact of the US Executive Order of May 2021 on the quality of SBOMs. Originally, the intention was to identify projects with SBOM files both before and after the Executive Order for a detailed historical analysis. However, due to the unavailability of individual projects fulfilling this condition, we decided to consider projects that have SBOMs before and after May 2021. Quasi-experimental designs are chosen when random assignment is impractical.

4.4.1 Variables and participants

- **Independent Variable:**The independent variable is the Executive Order, which creates two natural (e.g.,non- randomly assigned) variables which is time period categorized as "Before" the Executive Order and "After" the Executive Order, allowing a comparison of SBOM quality scores pre and post-intervention.
- **Dependent Variable:**The dependent variable is the quality score of SBOMs, assessed quantitatively using the SBOMQS tool.
- **Participants:**Software projects assessed using the SBOM Quality Scoring (SBOMQS) tool, representing a diverse range of domains and industries.

4.4.2 Implementation

Policy Impact Assessment: This phase examines the impact of the US Executive Order on SBOM quality among open source projects. SBOMs, both pre and post-implementation, are collected for comparative analysis. The focus is on changes in structure, completeness, and policy compliance, providing insights into the effect of governmental policy on SBOM practices and software supply chain security.

To answer RQ2, four projects were chosen from before the US Executive order and four projects from after the US Executive order. To understand the impact of the order created, the quality of the SBOM files themselves needs to be taken into consideration. The open-source SBOMqs tool provides automated quality benchmarking of SBOM files based on a comprehensive rubric encompassing over 30 validation checks across 10 categories of SBOM best practices.

Key quality evaluation dimensions include:

- Structural - Validity against SBOM data standards like CycloneDX and SPDX specifications.Assesses correct schema versions, supported file formats, and syntactic correctness.
- Completeness - Presence of minimum recommended SBOM data elements for component metadata like names, versions, licenses, and dependencies. Established by NTIA guidelines.
- Sharing Support - Inclusion of unencumbered usage rights and adherence to formats.
- Semantic Accuracy - Congruity between the SBOM dependency graph and real-world software optimized for tool consumption like CycloneDX XML composition. Confirms declared licenses, checksums, and relationships match reality.

SBOMqs (SBOM Quality Score) is an open-source command line tool created by Interlynk to systematically analyze and quantify the completeness, accuracy, and the overall quality of SBOM documents. It serves as a standardized benchmark capable of objectively scoring SBOMs based on how well they meet best practices criteria across several dimensions:

- Structural Quality: Assesses completeness of required SBOM structural elements like component names/versions.
- NTIA Minimum Elements: Checks for mandatory SBOM data fields defined by NTIA standards guidance.
- Semantic Quality: Measures accuracy of dependency relationships represented in the SBOM model.
- Sharing Quality: Evaluate the presence of data to support SBOM exchange and reuse.

The SBOM benchmark tool SBOMqs was used to evaluate the quality score of each SBOM file. Go was installed initially using the following script:

go install github.com/interlynk-io/SBOMqs@latest

This installed SBOMqs. Each SBOM file was then scanned with SBOMqs to get its quality score. The following script was used to generate the quality score for each file:

SBOMqs score <SBOM-file>

An example of how the result will look like after scanning an SBOM file with SBOMqs tool is shown here A.4

SBOM Quality Evaluation: This phase aims to evaluate the quality of the SBOM, particularly following the Executive Order. Various criteria are employed to assess the quality of SBOMs, such as the completeness of information, the accuracy of vulnerability disclosures, and the conformity with industry standards. The primary objective is to determine the effectiveness of the policy in enhancing the quality of SBOMs. Additionally, an Interrupted time-series analysis is carried out to understand the impact of the executive order.

Interrupted time series analysis

Interrupted Time Series Analysis, also referred to as quasi-experimental time series analysis, is a statistical method used to analyze the impact of an intervention by tracking data over an extended period before and after a specific point of intervention [39] [40]. The term time series denotes the continuous dataset over the observed period, and the interruption represents the controlled external influence or set of influences introduced during the intervention. This method involves systematically examining data patterns before and after the intervention to gain a comprehensive understanding of its effects. In our specific context, the intervention is the issuance of the US Executive Order in May 2021, which mandated SBOMs. We performed an ITS analysis on the quality ratings of eight software projects in order to assess the effect of a significant intervention. Statistical analysis was used to evaluate the quality scores of these projects, which are shown in Tables 5.6 and 5.7.

Justification: The Interrupted Time Series (ITS) analysis is a statistical method that is particularly well-suited for evaluating the impact of interventions over an extended period. This technique allows us to track and analyze data trends both before and after the implementation of the US Executive Order in May 2021. The process involves a few steps: first, preparing the data; next, using Ordinary Least Squares (OLS) regression for statistical modeling; then, conducting significance testing; and finally, interpreting the results. We selected OLS regression because it fits well with our dataset characteristics. Our analysis includes key variables like the dependent variable (Quality Score), independent variable (Time Period - Before/After the Executive Order), and a binary indicator variable (Time-Period), helping us study the periods before and after the intervention.

Hypotheses:

- Null Hypothesis (H0): Quality scores show no significant difference before and after the Executive Order.
- Alternative Hypothesis (H1): The quality scores significantly changed after the Executive Order.

We chose Ordinary Least Squares (OLS) regression as it is recommended for datasets with fewer than 12 points, aligning with our eight-project dataset [41]. Appendix A.5 shows the formula for the regression model.

4.5 Vulnerability scanning tool selection:

1. Why Vulnerability Scanning is Key: In today's connected world, software weaknesses called vulnerabilities are not just potential threats; they are real and present dangers. Every piece of code, every component, and every connection between them can have vulnerabilities that attackers can use to break into our computers and steal data. If we don't check for these vulnerabilities, the consequences can be serious, including data breaches, financial losses, and damage to our reputation. That's why vulnerability management is not a choice; it's an essential part of keeping our software safe and protecting our digital assets.

As software systems grow bigger and more complex, vulnerability scanning tools have become essential tools in the fight against cybersecurity threats. These tools automatically analyze SBOM data, which is a list of all the components in a software system, and compare it to a database of known vulnerabilities. This helps us find and fix vulnerabilities before attackers can exploit them. To effectively scan the extensive collection of SBOM files across 16 projects (8 Python and 8 Go), a thorough evaluation of three prominent vulnerability scanning tools was conducted: OSV Scanner, Snyk, and Grype. Each tool was assessed based on its capabilities, integration with SBOM formats, accuracy, and overall performance.

2. Comparing Three Leading Tools: To effectively scan the extensive collection of SBOM files across the 16 projects (8 Python and 8 Go), three prominent vulnerability scanning tools were compared: OSV Scanner, Snyk, and Grype. Each tool

was assessed based on its capabilities, integration with SBOM formats, accuracy, and overall performance

1. **OSV Scanner:**

   Strengths:

   - Comprehensive Coverage: OSV Scanner boasts an extensive database of known vulnerabilities, ensuring thorough analysis of the SBOM data.
   - SBOM Integration: OSV Scanner seamlessly integrates with various SBOM formats, including CycloneDX and SPDX, enabling seamless processing of the collected SBOM files.
   - Accuracy and Performance: OSV Scanner demonstrates high accuracy in vulnerability detection and maintains efficient performance even when handling large datasets.

   Limitations:

   - Primarily Focused on Containerized Applications: OSV Scanner is primarily designed for containerized applications, so its functionality may be limited for traditional software development.
   - CLI-Based Interface: OSV Scanner's CLI interface may require additional scripting for integration into complex workflows.

2. **Snyk:**

   Strengths:

   - Extensive Support: Snyk offers extensive support for various programming languages and frameworks, making it a versatile option for diverse projects.
   - User-Friendly Interface: Snyk's web interface provides a straightforward and intuitive experience for managing vulnerability scans and reports.
   - CI/CD Integration: Snyk integrates seamlessly into CI/CD pipelines, enabling automated scanning and remediation processes.

   Limitations:

   - Premium Model: Snyk operates on a Premium model, with limited functionality for free users, requiring a paid subscription for advanced features.
   - SBOM Scanning Configuration: SBOM scanning with Snyk may require additional configuration to ensure thorough coverage.

3. **Grype:**

   Strengths:

   - Lightweight and Efficient: Grype is a lightweight and efficient tool, making it suitable for scanning large volumes of SBOM data.

- CI/CD Integration: Grype integrates easily into CI/CD pipelines, ensuring automated vulnerability detection and remediation.
- Source Code and SBOM Scanning: Grype supports scanning both source code and SBOMs, providing a comprehensive vulnerability assessment.

Limitations:

- Smaller Vulnerability Database: Grype's vulnerability database is smaller compared to OSV Scanner and Snyk, potentially leading to missed vulnerabilities.
- In-depth Configuration: For in-depth analysis, Grype may require additional configuration beyond its default settings.

3. Choosing OSV Scanner: After carefully considering the strengths and limitations of each tool, OSV Scanner was ultimately selected as the preferred vulnerability scanning tool for this project. This decision was driven by the following factors:

- Comprehensive Vulnerability Coverage: OSV Scanner's extensive database of known vulnerabilities ensures thorough analysis of the SBOM data for a comprehensive threat profile.
- Seamless SBOM Integration: OSV Scanner's seamless integration with various SBOM formats enables straightforward processing of the collected SBOM files without additional data transformation. High Accuracy and Performance: OSV Scanner demonstrates high accuracy in vulnerability detection and maintains efficient performance even when handling large datasets.
- Open-Source and Freely Available: OSV Scanner's open-source nature eliminates licensing costs and ensures accessibility for any project.

While Snyk and Grype offer valuable features and capabilities, OSV Scanner's comprehensive vulnerability coverage, seamless SBOM integration, high accuracy, and open-source nature made it the most suitable choice for this project's specific requirements.

Vulnerability scanning: The SBOM files collected from the archival study are used to perform vulnerability scanning and identify potential security weaknesses within the software ecosystem. To achieve comprehensive vulnerability detection, a two-pronged approach is employed, utilizing both the OSV Scanner tool and the OSV API. Here you find the GitHub URL of OSV-scanner https://github.com/google/osv-scanner

Initial Scanning with OSV Scanner

The first step in the vulnerability scanning process involved installing the OSV Scanner tool using the Scoop package manager. Scoop is a lightweight and user-friendly command-line tool for managing software packages on Windows systems. To install OSV Scanner using Scoop, the following command was executed in Windows PowerShell:

PowerShell
scoop install OSV-scanner
Once the installation was complete, the collected SBOM files were scanned using the OSV Scanner tool. The command used to scan each SBOM file was:

PowerShell
OSV-scanner –SBOM <file-path>
In this command, <file-path> represents the absolute path to the SBOM file being scanned. This command effectively scanned the SBOM file and generated a list of identified vulnerabilities displayed to the console. However, for further analysis and archival purposes, the following command was employed to generate a JSON file containing the scanning results.

PowerShell
OSV-scanner –SBOM <file-path> -json > output.json
This command generated a JSON file named output.json in the current directory, containing a structured representation of the vulnerabilities identified during the scan. An example of how the result will look like when scanned with OSV scanner is shown here A.3

Addressing Scanning Limitations with OSV API

While the initial scanning approach using the OSV Scanner tool yielded satisfactory results, certain files remained inaccessible to the tool. To address this limitation and ensure comprehensive vulnerability assessment, the OSV API was leveraged. The OSV API provides a programmatic interface for accessing the OSV vulnerability database, enabling querying vulnerability information for specific packages.

To utilize the OSV API, the SBOM file was first uploaded to Google Colaboratory, a cloud-based platform for interactive coding. The content of the uploaded file was then extracted and converted into a string format. Next, the URL for the OSV API, " https://github.com/google/OSV.dev", was defined. An empty dictionary was initialized to store the scanning results. A regular expression pattern was defined to extract package names from the SBOM content. Using this pattern, all package names were extracted from the SBOM content. To achieve a comprehensive vulnerability assessment, an iterative process was implemented where API requests were made for each extracted package name. For each API request, a payload containing the package name and its corresponding ecosystem was constructed. These API requests were then sent to the defined API URL using the prepared payloads. The JSON response from each API request was parsed and stored in the initialized results dictionary. Additionally, all extracted package names were printed to the console for verification.

Analysis

We have collected all the SBOM files from the archival study along with their corresponding vulnerability scanning result files. Now, we need to analyze the total data thoroughly.

This analysis will involve four distinct phases: Evolution Analysis, Vulnerability Analysis, Comparison of Python and Go Ecosystems, and SBOM Analysis by Criteria. The Evolution Analysis Phase will carefully review historical SBOM data to uncover trends and patterns in how vulnerabilities are introduced and fixed. The Vulnerability Analysis Phase will dig deeper into the nature and severity of identified vulnerabilities, allowing for effective prioritization and the development of practical remediation strategies. The Comparison of Python and Go Ecosystems Phase will reveal any significant differences in vulnerability profiles between applications built on these two programming languages.

During the SBOM Analysis by Criteria Phase, we will analyze the critical vulnerabilities found in the projects. This analysis will be based on the CVSS scores and CWE IDs. The CVSS vector will be used to evaluate the CVSS score of the vulnerabilities, and the CVSS calculator will be available online to evaluate the quality scores. The most common and repeated CWE IDs found in the projects will be cross-referenced with the CWE-ID website to explore the vulnerabilities and their remediation techniques. To access the CVSS calculator 3.1 you can use the following URL:
https://www.first.org/cvss/calculator/3.1

Chapter 5

Results and Analysis

This section gives an overview of the results obtained and analyses of those results in this thesis work.

5.1 RQ1 - What information contained in SBOMs

Header Type	**Description**	**Significance**	**Example**
Unique SPDXID	Unique identifiers like 'SPDXRef-DOCUMENT'	Crucial for tracking and referencing SBOMs.	SPDXRef-DOCUMENT-001
Name	Specifies the relative path to the SBOM file as "."	Identifies the SBOM file location.	./path/to/SBOM file
SPDX Version	Lists version of SPDX format used - "SPDX-2.2"	Indicates the SPDX format version used.	SPDX-2.2
Creation Info	Timestamp and creator details.	Captures when and by whom the SBOM was created.	2023-12-14, John Doe
Data License	License for SBOM data distribution as "CC0-1.0".	Specifies the license covering SBOM data distribution.	CC0-1.0
Document Namespace	A unique URI identifying the SBOM document.	Enables retrieval and identification of the SBOM.	https://example.com/SBOM

Table 5.1: Information on SBOM file(Header section)

Package Detail	**Description**	**Significance**	**Example**
Package Identifiers	Unique IDs assigned to each package for tracking.	Crucial for tracking across SBOMs.	SPDXRef-56da0120c8f31bb0
Package Names	Names that provide insights into the dependency tree.	Aids in vulnerability assessment and management.	cloud.google.com/go
Versions	The exact package version in use, when detectable.	Allows mapping vulnerabilities to affected version ranges.	v1.18.2
Licenses Data	Applicable license metadata.	Some may state "NONE," highlighting detection difficulties.	MIT License
External References	Links to package manager security data and other sources.	Enables further research and validation of package security.	[Link to security data]
Download Locations	Direct package download locations.	"NOASSERTION" may indicate availability uncertainty.	[Link to download location]

Table 5.2: Information on SBOM file(Package details)

Tables 5.1 and 5.2 provide information that should be present in an SBOM file. Typically, an SBOM file consists of two sections - the header section and the package details section. The header section contains crucial metadata for tracking and referencing SBOMs. It provides essential details about the SBOM itself, including a unique identifier (SPDXID/CycloneDXID), name, version (SPDX/cycloneDX Version), creation information (timestamp and creator), data license, and a unique namespace for identification and retrieval. After the header section, you will find the package details section. This section provides more information about each package listed within the SBOM. Each package has a unique identifier and a descriptive name, along with specific versions used (if detectable). Package identifiers are crucial for cross-SBOM tracking, while package names help to understand the dependency trees, making vulnerability assessment and management easier. Including version details allows mapping vulnerabilities to affected version ranges, and license data highlights potential challenges in detection. License data plays a crucial role in indicating

open-source licenses or highlighting challenges in detection. External references and download locations enable further research and validation of package security.

5.2 Information in vulnerability scanning result file:

Section	Element	Description	Example
1	Vulnerability		
	Schema Version	The structured data format used for the advisory record.	1.6.0
	ID	Uniquely identifies the vulnerability.	GO-2022-0646
	Aliases	Alternate reference IDs typically from other databases.	CVE-2020-8911
2	Publication Details		
	Published	The date when the vulnerability was first publicly disclosed.	2022-02-11T23:26:26Z
	Modified	Date the advisory record was last updated.	2023-06-12T18:45:41Z
	nvd published at	Date incorporated into NVD government database.	
3	Descriptions		
	Summary	Single line synopsis of the flaw identified.	Use of risky cryptographic algorithm in github.com/aws/aws-sdk-go
	Details	Multi-line analysis of impact, patches, workarounds, references, and credits.	

	Severity > Score	Standard severity rating systems.	CVSS vector - CVSS:3.1/AV:L/AC:L /PR:L/UI:N/S:U /C:N/I:N/A:H
4	Package Details		
	Package	Namespace, name, and package manager identifier string.	pkg:golang/github.com /aws/aws-sdk-go
	External References > purl	Related package URL.	pkg:golang/github.com /aws/aws-sdk-go@v1.42.24
5	Version Ranges Affected		
	Imports > path	File location containing the vulnerability.	github.com/aws/aws-sdk-go/service/s3/s3crypto
	Imports > symbols	Exposed functions enabling exploit.	NewDecryptionClient
6	References		
	Type	Categorization as advisory/fix instructions.	ADVISORY
	URL	Link/URI to external reference.	[https://aws.amazon .com/blogs/developer/ updates-to-the-amazon-s3-encryption-client/?s=09](https://aws.amazon.com/blogs/developer/updates-to-the-amazon-s3-encryption-client/?s=09)
7	Severity Details		
	Type	Standard rating system used (e.g., CVSS).	CVSS
	Score	Actual vector or numeric score assigned.	CVSS v3 vector - CVSS:3.1/AV:L/AC:L /PR:L/UI:N/S:U/C:N /I:N/A:H

8	Database Specific Details		
	cwe_ids	Related Common Weakness Enumeration (CWE) IDs.	CWE-400
	github_reviewed	Boolean indicating if GitHub reviewed vulnerability.	true
	github_reviewed_at	Date GitHub review completed if applicable.	2022-06-06T22:07:10Z
9	References		
	Type	Classification as advisory, patch, etc.	ADVISORY
	URL	Link/URI for reference.	https://example.com/reference
10	Affected Versions		
	Package name and language ecosystem		
	Version range tuples with type, introduced/fixed semantics		type: SEMVER, introduced: 0, fixed: 1.5.13

Table 5.3: Detailed Breakdown of the Vulnerability result file

The vulnerability scanning result file, which is presented in Table 5.3, is created by scanning for potential vulnerabilities within an SBOM file. This file provides a detailed record that includes critical information related to vulnerabilities. It begins with the identification schema version and a unique vulnerability ID and then expands to include aliases, publication details, and modification timestamps. The descriptions offer a concise summary and a detailed analysis of identified flaws, while severity details provide standard rating systems and scores for prioritization. Package details provide essential information such as namespace, name, and manager identifier, which are complemented by external references. The file provides information about affected version ranges, file locations, and important symbols for understanding potential exploits. It also includes references that categorize fix instructions and advisories, along with URLs for external resources. The severity details provide insights into the type and severity scores (CVSS scores), related Common Weakness Enumeration (CWE) IDs, GitHub review status, and completion dates. This breakdown can help companies make smarter decisions and come up with effective solutions to improve their cybersecurity, making their entire security system stronger.

5.3 RQ1.1 - Evolving over time

5.3.1 Number of version changes over time:

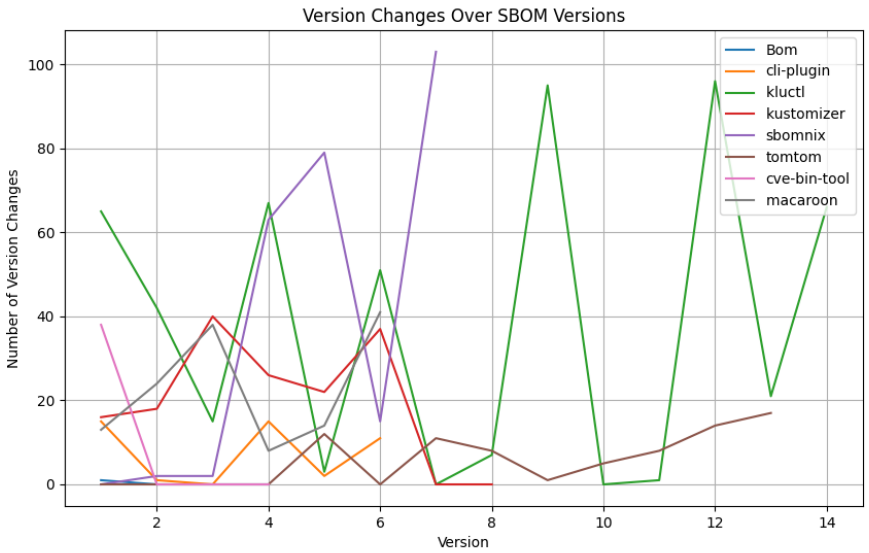


Figure 5.1: Version Changes over time

The figure 5.1 above depicts the number of version changes that occurred over time in the SBOM file for certain projects. The results reveal that the number of version changes increased for most projects over time. The graph provides information about the version changes of 8 different projects from both Python and Go ecosystems. It shows that as versions increased, the count of version changes also increased. For instance, in the early version of the SBOMnix project, the number of version changes was zero, while in its latest version, it increased to above 100. Similarly, in the case of the macaroon project, there is an increase in the number of version changes in the latest versions. While some projects like Bom and Cve bin tool did not see any license changes in their latest versions, and the kluctl project's version changes were variable across its versions, but the count has increased in their latest version. Overall, the graph communicates that the number of version changes has been increasing over time for the projects.

5.3.2 Number of removed and added packages:

Figures 5.2 and 5.3 depict the changes in the number of removed and added packages across different versions of the SBOM. As shown in Figure 5.2, most projects experienced a minor increase in the number of removed packages over different versions. For instance, projects such as BOM, CLI plugin, and Kustomizer had a count of 100 removed packages across versions, such as from BOM version 0 to 3 or from CLI plugin version 3 to 4. However, one project exhibited a significant increase in the number of packages removed in its latest version, with a count of over 1000. This suggests that newer versions of SBOM are more likely to remove packages than older

versions. There are several possible reasons for this trend. One possibility is that newer versions of SBOM are more accurate and can identify more packages that are no longer necessary. Alternatively, newer versions of SBOM might have stricter requirements, which lead to the removal of packages that do not meet those standards. do not meet those standards. The increase in the number of removed packages may be due to changes in the way that the SBOM is generated. For instance, if a different tool is being used to generate the SBOM, it may remove packages more often. Additionally, in Figure 5.3, we can observe a rise in the number of newly added packages across different versions of the SBOM. For example, the latest versions of cli plugin and kustumzier have added more packages. Moreover, projects like SBOMnix and Kluctl show a steady increase in the count of newly added packages over versions. For SBOMnix, the count goes up to 700 and for Kluctl, it goes to more than 1400.

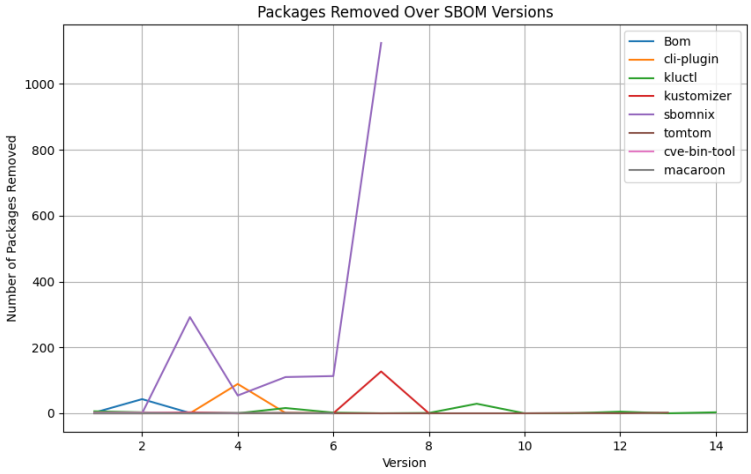


Figure 5.2: Packages removed over SBOM versions

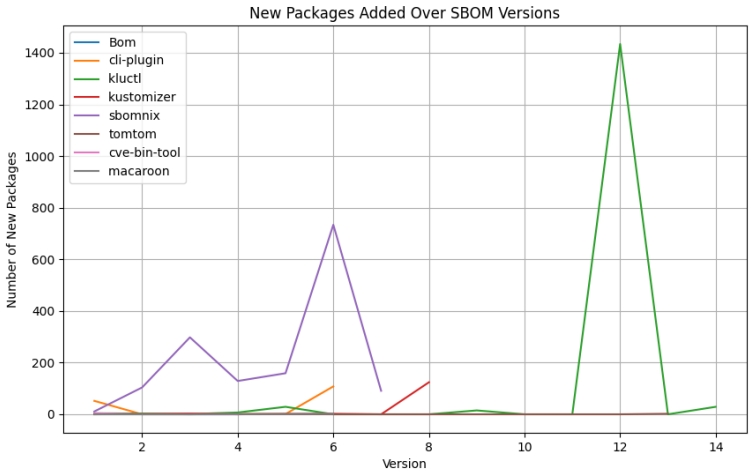


Figure 5.3: New Packages added over SBOM versions

5.3.3 Number of License changes over time:

The figure 5.4 below shows the number of license changes made over SBOM versions. For projects like SBOMnix, the number of license changes has increased to more than 200 from version 2 to 4, and for Kluctl, it has increased to 247 from version 10 to 12. Overall, the graph indicates that there has been a rising trend in the number of license changes made to the SBOM over time. This trend could be due to various factors, such as changes in the SBOM itself, modifications in the licensing requirements of the components included in the SBOM, or changes in the way license changes are tracked.

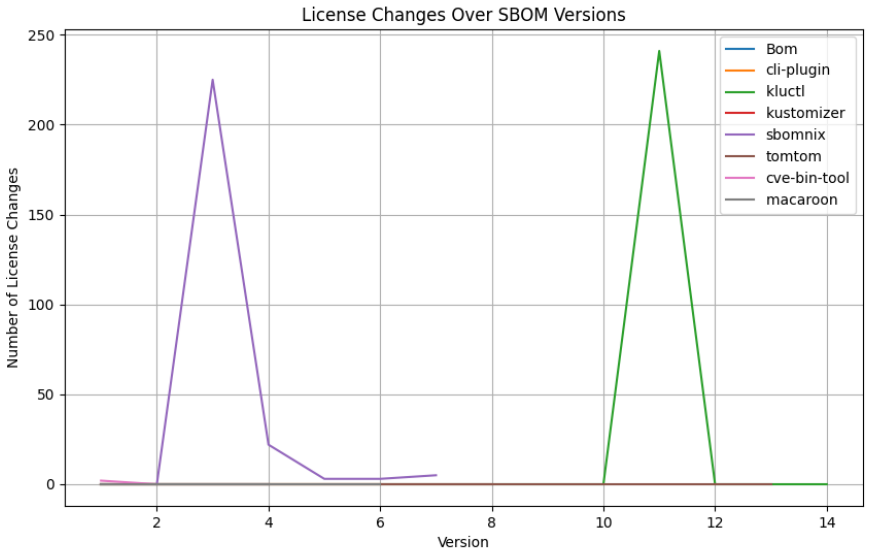


Figure 5.4: Number of License changes over SBOM versions

5.3.4 Number of components:

The Python ecosystem had SBOMs from 9 different projects, with the number of components ranging from 7 to 81 across different versions. Overall, the Python projects showcased a trend of increasing number of components over time, indicating growth and evolution in codebases. Specific projects like Tomtom and SBOMnix showed significant increases in components between some versions, such as 48 to 49 for Tomtom and 0 to 1525 for SBOMnix. Similarly, the Go ecosystem had SBOMs from 6 different projects, with the number of components ranging from 46 to 1781 across versions. Most Go projects exhibited relatively stable numbers of components across versions, with some gradual increases over time. Notable increases were seen in bom (131 to 180) and kluctl (213 to 1751), indicating sizable changes.

TSA results for components:

Table 5.4 below shows the time series analysis results of component growth in Python and Go projects

Feature	Python	Go
Total projects	9	6
Versions analyzed	38	54
Minimum components	7	46
Maximum components	81	1781
Average components per project	48.8	420.5
Median components per project	48	248
Average growth per version	2.3	23.6
Largest growth in a version	1525 (SBOMnix 0.2.0)	1538 (kluctl 2.21.0)
Min Slope	0.15	1-0.66
Max Slope	1.93	123.63
Mean Slope	1.04	4.75

Table 5.4: TSA of Component Growth in Python and Go Projects

On average, Go projects have significantly more components (421) than Python projects (49). Python projects tend to grow at a slower pace compared to Go with an average of 2.3 more components per release, while Go projects show an average increase of 24 more components per release. However, both ecosystems have seen outlier cases where over 1,500 more components were added in a single release. The median growth rates for both ecosystems show that Go projects have bigger codebases and increments with an average of 248 new components per release, while Python projects have an average of 48 new components per release. The largest single version change for both Python (SBOMnix) and Go (kluctl) saw over 1500 new components added, but these seem to be outliers. The significantly higher average number of components in Go projects (421 vs 49 for Python) quantitatively demonstrates Go systems have larger codebases. Python's small average component growth per release of 2.3 reflects incremental code changes over time rather than major restructuring Go's higher average growth of 24 more components per release indicates less stability between versions.

5.3.5 Licenses:

Python Projects: Python projects typically have fewer licenses, with most having 0 to 7 licenses. The average number of licenses used in Python is only 3.5. This suggests a preference for using compatible licenses. This could be because Python has a larger ecosystem of libraries and tools, which makes it easier to find compatible licenses.

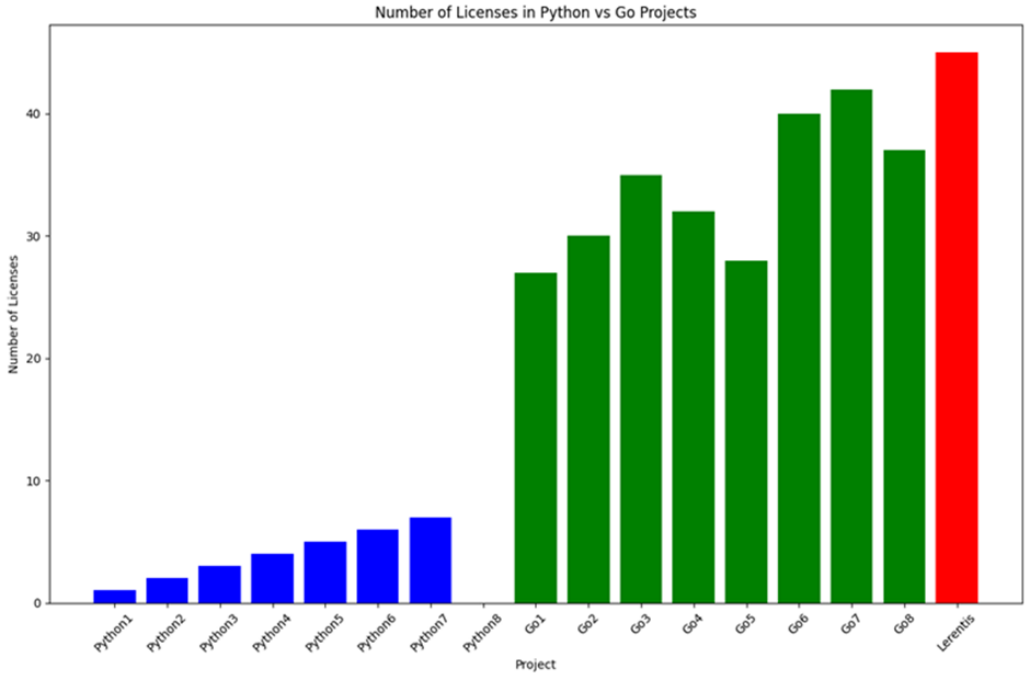


Figure 5.5: Number of Licenses in Python vs GO

Go Projects: Go projects consistently report a higher number of licenses, surpassing 27 distinct licenses. This diversity implies a broader utilization of dependencies, reflecting Go's flexibility or different development practices. This could be due to the fact that Go is a newer language with a smaller ecosystem of libraries and tools. As a result, Go projects may be more likely to use a wider range of licenses, including private licenses. Special Attention: Lerentis Go Project: With 45 licenses, the Lerentis Go project stands out and illustrates the difficult issues associated with maintaining legal compliance in technology architectures that depend on a wide range of interdependencies. Overall, the analysis shows that there is a significant difference in the number of licenses used by Python and Go projects. Python projects typically use fewer licenses, while Go projects typically use a higher number of licenses. This difference could be due to a number of factors, including the size and maturity of the respective ecosystems, the licensing terms of the libraries and tools used, and the development practices of the respective communities.

5.3.6 Vulnerability Trends Over Time

The table 5.5 displays the results of the comparison of vulnerabilities found in different projects. In figures 5.6 to 5.11, line graphs depicting the number of vulnerabilities over time for each project are presented. Bar graphs 5.12 and 5.13 illustrate the comparison of vulnerabilities and the percentage reduction in vulnerabilities for each project in a visual manner.

Project	Initial Vulnerabilities	Latest Vulnerabilities	Percentage Reduction
SBOMnix	29	0	100
Flux	55	4	90.91
bom	48	12	75
cli-plugin	37	1	97.30
Tomtom	20	20	0
cve-bin-tool	2	2	0

Table 5.5: Comparison of Vulnerabilities across projects

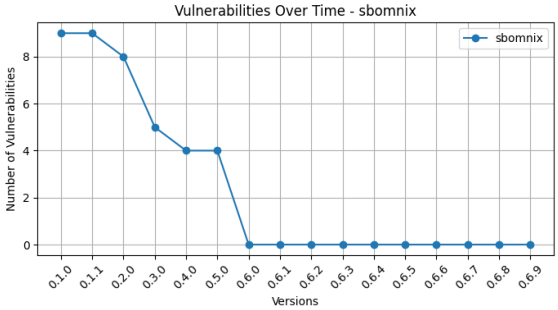


Figure 5.6: sbomnix vulnerabilities

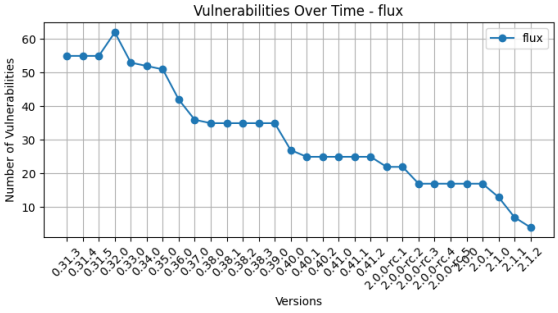


Figure 5.7: Flux vulnerabilities

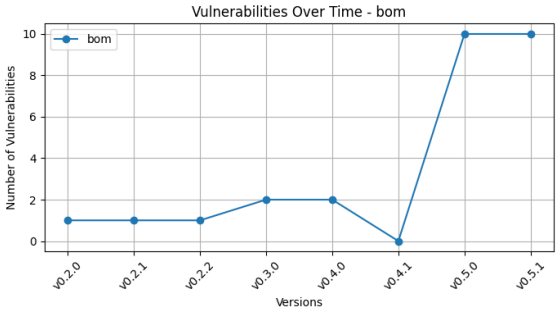


Figure 5.8: bom vulnerabilities

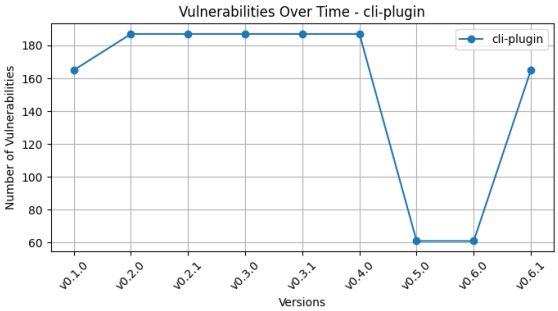


Figure 5.9: cli-plugin vulnerabilities

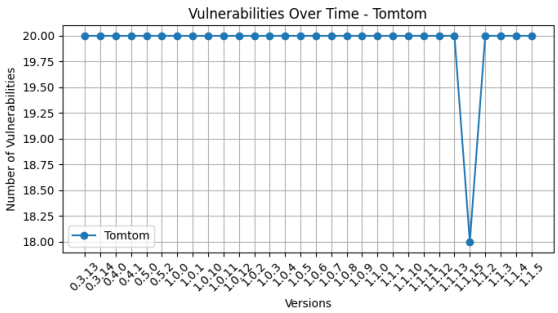


Figure 5.10: tomtom vulnerabilities

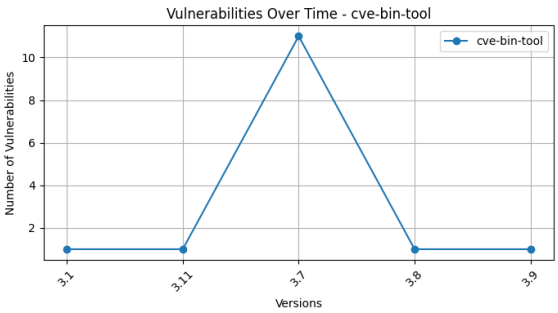


Figure 5.11: cve-bin-tool vulnerabilities

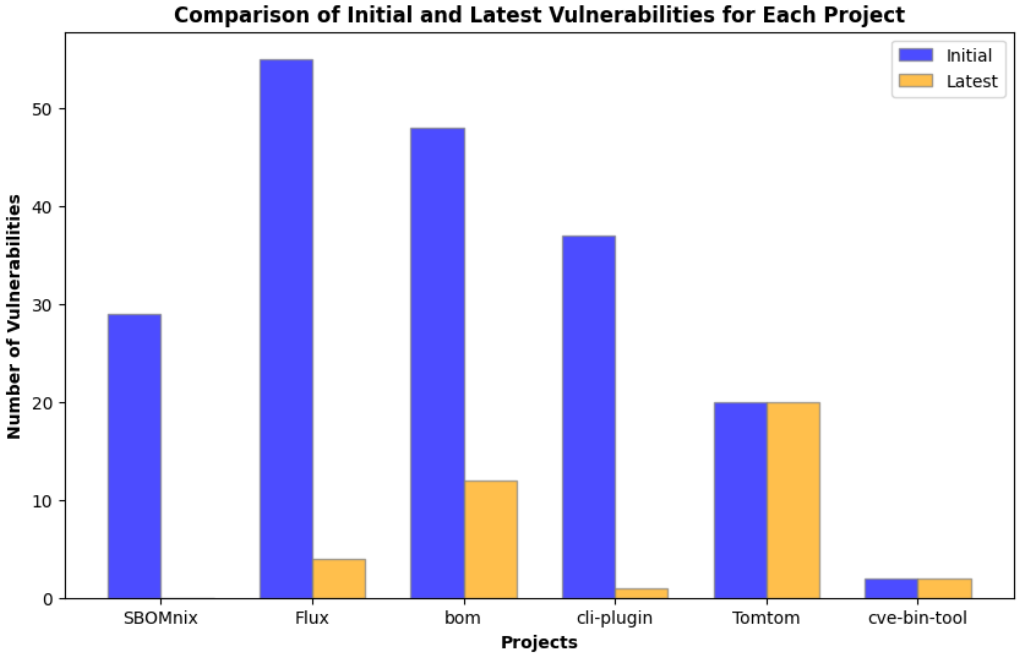


Figure 5.12: vulnerability-comparison-plot

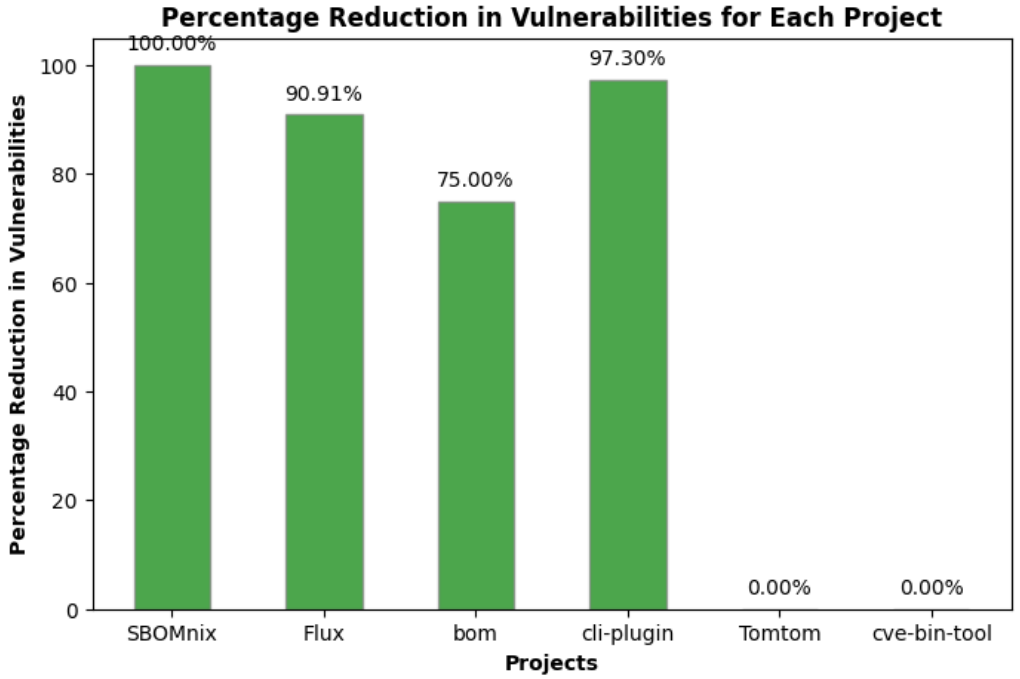


Figure 5.13: percentage-reduction-plot

The Sbomnix project initially had the highest number of vulnerabilities at 29 (Table 5.5). However, its vulnerability trend graph (Figure 5.6) depicts a consistent and steady decline, which corresponds to the 100% reduction in vulnerabilities reported in the table 5.5‘. The graph shows that by the end of the observed period, there were no vulnerabilities left in the project.

Similarly, the Flux project had 55 initial vulnerabilities. Its vulnerability graph (Figure 5.7) also shows a decreasing trend, but with some fluctuations along the way. These fluctuations suggest that there were some challenges or setbacks in the vulnerability remediation process. Nevertheless, the graph supports the 90.91% reduction in vulnerabilities reported in Table 5.5. The vulnerability trend graphs of the bom, cjs-plugin, tomtom, and cve-bin-tool projects are presented in Figures 5.8, 5.9, 5.10, and 5.11 respectively. The graphs for bom and cli-plugin projects show a reduction in vulnerabilities. The bom project's graph reflects a 75% reduction in vulnerabilities, with an initial decline followed by a spike and ultimately a decrease to around 12 vulnerabilities. This aligns with the vulnerabilities reported in Table 5.5. The cli-plugin project's graph displays an irregular pattern with spikes and drops, but ends with around 1 vulnerability, consistent with the table's data of a 97.30% reduction and 1 latest vulnerability. The tomtom project's vulnerability trend graph shows a relatively stable trend with a sudden spike towards the end, matching the table's data where no reduction in vulnerabilities was observed. Finally, the cve-bin-tool project's vulnerability graph remains constant at around 2 vulnerabilities throughout the observed period, aligning with the table's data of no reduction and 2 latest vulnerabilities. The analysis indicates that numerous open-source software projects tend to begin with a high number of vulnerabilities but improve their security over time by addressing vulnerabilities in later versions. Therefore, many OSS projects can significantly decrease their vulnerabilities and enhance their security over time. This emphasizes the significance of developers maintaining and updating their software because the latest versions are generally more secure. However, certain projects continue to have constant vulnerabilities (such as Tomtom, which has around 20 vulnerabilities per release) or remain low throughout (like cve-bin-tool). Therefore, although security often improves, the progress depends on the specific project. In summary, the SBOM scan data analysis over time demonstrates that many open-source projects can effectively reduce their vulnerabilities in newer versions. But maintenance varies across projects, emphasizing the necessity for continued security best practices when updating and securing both older and newer software releases.

5.4 RQ2 - Impact of US Executive Order

Table 5.6 shows the quality assessment results pre-intervention of the order and table 5.7 shows the quality assessment post-intervention of the order.

Project	**STRUC-TURAL**	**NTIA-MINIMUM-ELEMENTS**	**SEMAN-TIC**	**SHARING QUALITY**	**SCORE**
Fossology rest-api	10	7.1	10	2.9	6.8
Internet standards	10	7.1	3.3	4.1	6.3
Pystacks	10	5.7	6.7	4.3	6.4
Boyter	10	5.7	6.7	4.3	6.4

Table 5.6: Quality Assessment Pre-Intervention of the Order

Project	**STRUC-TURAL**	**NTIA-MINIMUM-ELEMENTS**	**SEMAN-TIC**	**SHARING QUALITY**	**SCORE**
cve-bin-tool	10	9.9	6.2	9.7	9.4
Lerintiz	10	9.9	6.3	7.9	8.8
bom	10	8.5	6.4	7	8.1
Kluctl	10	8.6	3.3	4.3	6.8

Table 5.7: Quality Assessment Post-Intervention of the Order

Feature	**Before**	**After**
Structural	10.000	10.000
NTIA Minimum Elements	6.400	9.225
Semantic Quality	6.675	5.550
Sharing Quality	3.900	7.225
Score	6.475	8.275

Table 5.8: Comparison of Feature Means Before and After Improvements

Feature	Std Before	Std After
Structural	0.000000	0.000000
NTIA Minimum Elements	0.808290	0.780491
Semantic	2.735416	1.502221
Quality	0.673300	2.250000
Sharing Quality	0.000000	0.000000
Score	0.221736	1.117661

Table 5.9: Standard Deviation Comparison Before and After

ITS OLS Regression Results
Model Summary:

- R-squared: The model explains approximately 84.7% of the variability in quality scores.
- Adj. R-squared: The adjusted R-squared, which considers the number of predictors in the model, is 78.5%. It penalizes for adding less informative variables.
- F-statistic: The overall significance of the model is tested by the F-statistic, which is 13.80 in this case. The associated p-value is 0.00921, indicating that the model is statistically significant.
- Coefficients Constant (Intercept): The baseline quality score, when both Time and Time-Period are zero, is 7.675.
- Time: For each additional unit of time, there is an average decrease of 0.480 in the quality score. The p-value associated with Time is 0.043, suggesting statistical significance with an alpha-level of 5%.
- Time-Period: Post-intervention, the quality score increases by 3.720. The p-value associated with the time period is 0.006, indicating statistical significance with an alpha-level of 5%.

Appendix A.6 reports the detailed results after running Interrupted time series analysis.

5.4.1 ITS Resuts Analysis:

The data compares the quality scores of 4 software projects SBOMs before and 4 projects SBOMs after the May 2021 US Executive Order mandate to provide SBOMs. Several metrics were used to assess SBOM quality: structural, NTIA minimum elements, semantic quality, and sharing quality. Looking at the mean quality scores, there is an improvement in quality from before (6.475) to after (8.275) the Executive Order. After the changes, there was a noticeable increase in both the NTIA minimum elements score, from 6.400 to 9.225, and the semantic quality score, from 3.900

to 7.225. This indicates the Executive Order is associated with improved compliance with NTIA guidelines and improved semantic understandability of the SBOMs. The standard deviations also declined after the Executive Order for the NTIA, semantic, and overall scores, showing less variability and more consistency in meeting quality benchmarks. Specifically, the standard deviation of the NTIA minimum elements score decreased from 0.808 before the order to 0.780 after. This shows that in addition to the 44% increase in the mean score, there was less variability in coverage of important SBOM metadata elements following the enforcement of the mandates. Similarly, the standard deviation for the semantic quality score dropped more substantially from 2.735 before to 1.502 after. So not only did organizations improve the representation of dependencies by an average of 85%, but the degree of variability between SBOMs on this measure decreased notably as well under the new federal guidelines. Finally, the overall quality score standard deviation fell from 0.222 before to 1.118 after, along with a 28% gain in the average score. This set of metrics demonstrates a decrease in variability as well as an improvement in quality, as evidenced by the increased mean score and decreased variation. The model explaining about 84.7% of the variability in SBOM quality scores, which is indicated by the high R-squared value. Moreover, the adjusted R-squared value of 78.5% shows that the model is solid and avoids overfitting by taking out less important variables. The F-statistic of 13.80 is significant and has a p-value of 0.00921, which gives us more confidence in the overall statistical significance of the model. Importantly, the positive coefficient for the time period indicates a substantial post-intervention improvement, with the quality score increasing by 3.720 units. The findings underscore the effectiveness of regulatory interventions in driving improvements in the transparency and security of software components, as reflected in the quality metrics of SBOMs. In summary, analysis of the quality scores provides evidence that the May 2021 US Executive Order has had a positive impact on improving the quality of SBOMs produced. The mandate to provide SBOMs has been associated with SBOMs that better comply with expected guidelines and provide more complete, consistent and understandable information about software components.

5.5 RQ3 - CVSS scores and CWE

5.5.1 Data Collection and Organization

For each of the selected projects, SBOM files and vulnerability results from the OSV scanner were collected. The analysis aimed to identify critical vulnerabilities, resulting in the creation of a comprehensive spreadsheet that included the following key columns:

- Project Name
- Ecosystem
- Version
- File ID

- Criticality
- CVSS Score
- CWE ID
- CVSS Calculator Score

5.5.2 Identification of Critical Vulnerabilities

Critical vulnerabilities were prioritized in the analysis, considering their potential severity and impact on software security.

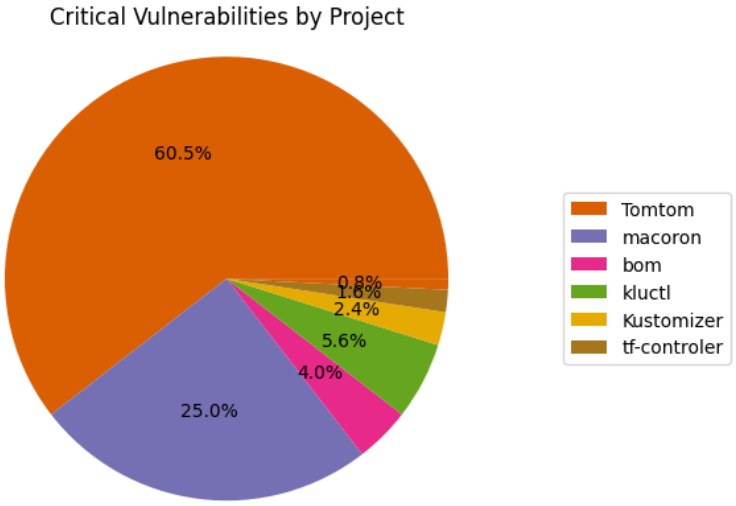


Figure 5.14: Distribution of total critical vulnerabilities across projects.

The pie chart shows the distribution of total critical vulnerabilities across 6 software projects. The project with the most critical vulnerabilities by far is Tomtom, accounting for 75% of the total critical vulnerabilities. This indicates that Tomtom likely has widespread code quality and security issues that allow for many critical exploits. After Tomtom, macoron has the second-highest number of critical vulnerabilities at 31. However, this is less than half of Tomtom's total.The remaining 9 projects have relatively few critical vulnerabilities, ranging from 0 to 7. SBOMnix, lerentis-bitwarden-crd-operator, cli-plugin, and cve-bin-tool have reported 0 critical vulnerabilities. Over 90% of the reported critical vulnerabilities are concentrated in just two projects - Tomtom and macoron.

5.5.3 Vulnerability Distribution Analysis

A comparative analysis across different software versions was conducted, highlighting the evolution of vulnerabilities in various projects.

The figure 5.7 above presents the comparison of vulnerabilities within four different software projects—SBOMnix, Macron, Bom, and cli-plugin—across various

Figure 5.15: Vulnerability distribution across different software versions.

released versions. Each subplot is dedicated to a specific project and tracks the presence and intensity of critical vulnerabilities (for SBOMnix, Macron, and Bom) and the total vulnerabilities (for cli-plugin) as they evolve from one version to the next.

SBOMnix Critical Vulnerabilities (Top-Left Subplot): This chart exhibits a significant decrease in critical vulnerabilities from version 0.1.0 to 0.2.0, with the count dropping rapidly to near zero. The subsequent versions, extending to 0.6.9, demonstrate a maintained baseline without the detection of critical vulnerabilities.

Macron Critical Vulnerabilities (Top-Right Subplot): This graph illustrates a variable pattern, where initial versions (0.1.0 and 0.2.0) are completely free of critical vulnerabilities. However, a notable increase occurs, peaking at version 0.5.0 with approximately 13 critical vulnerabilities, before descending in the subsequent version.

Bom Critical Vulnerabilities (Bottom-Left Subplot): The trajectory of critical vulnerabilities here shows a gradual increase, peaking in version v0.5.0, where the count reaches four. The y-axis scale offers a detailed view of the vulnerability distribution, emphasizing even small changes between versions, which is crucial for assessing the impact of version updates on security.

cli-plugin Total Vulnerabilities (Bottom-Right Subplot): The final chart highlights the total vulnerabilities for the cli-plugin component, characterized by significant volatility. There's a huge difference between versions v0.3.1 and v0.4.0. The number of vulnerabilities goes down at first and then goes up to a peak, showing a major change in vulnerability count.

5.5.4 Python vs Go

Python projects consistently show higher numbers of critical vulnerabilities (average of 14.9 per version) versus Go projects (average 3.6 per version) as shown in Figure 5.8. While earlier Python versions all had around 20 critical vulnerabilities and Go had 0-2, the latest Python version has 2, and the latest Go has 4 critical vulnerabilities as shown in below figure 5.9 Many Python project versions have around 20

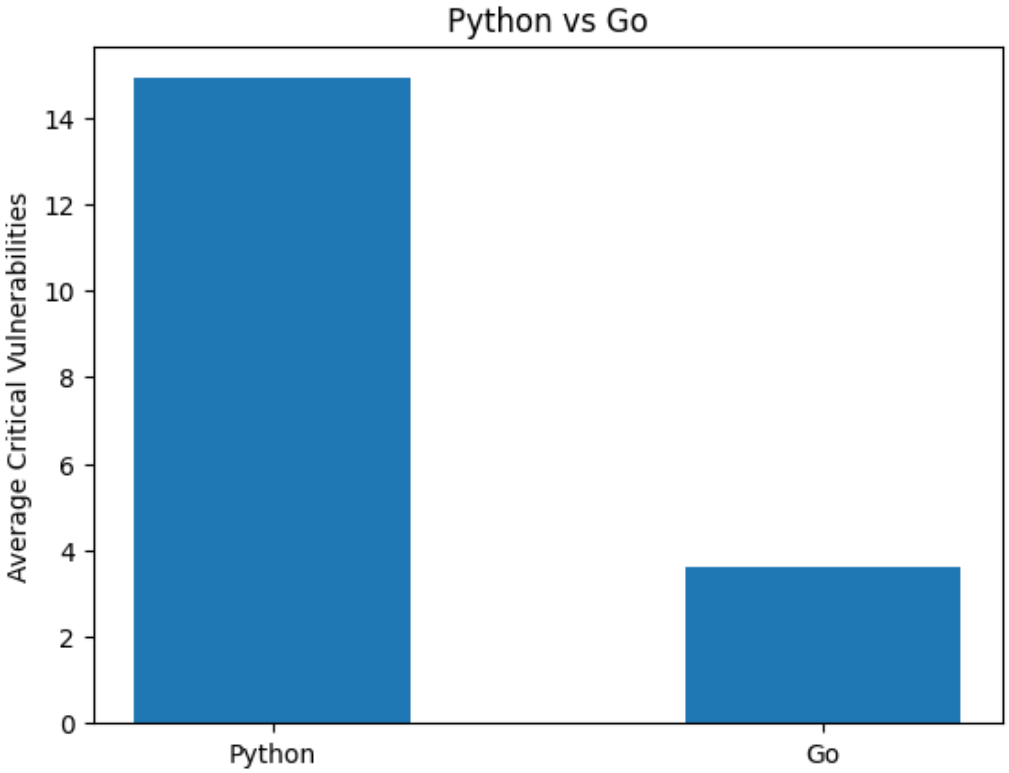


Figure 5.16: Python vs GO critical vulnerabilities

critical vulnerabilities, indicating major clusters with high counts. The variability in several critical vulnerabilities over versions is lower for Python. While Go projects had 0-2 critical vulnerabilities in early versions, later versions consistently showed 3-10 critical vulnerabilities as shown in Figure 5.10. So new Go versions tend to add more critical issues

5.5.5 CVSS Scores

CVSS scores were employed to quantitatively assess the severity of vulnerabilities identified in the selected projects from the Python and Go ecosystems. The calculated scores ranged from [9.1] to [9.8], providing a comprehensive overview of the severity levels.

The top four vulnerabilities, based on their calculated CVSS scores, are as follows:

5.5.6 CWE ID

An in-depth analysis of the identified vulnerabilities involved an exploration of the Common Weakness Enumeration (CWE) IDs. This step aimed to understand the specific characteristics and nature of each vulnerability. The CWE IDs were cross-referenced with the CWE website to extract detailed information about the vulnerabilities, providing qualitative insights.The following section summarize important information from the CWE website for each identified CWE ID:

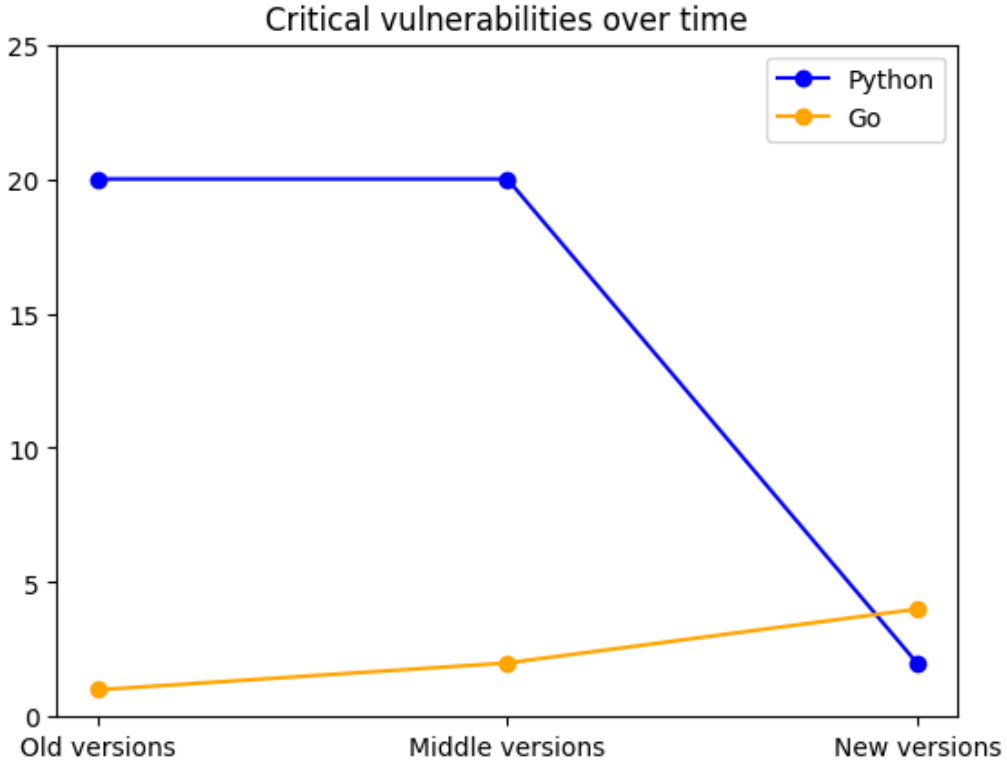


Figure 5.17: Python vs GO critical vulnerabilities over time

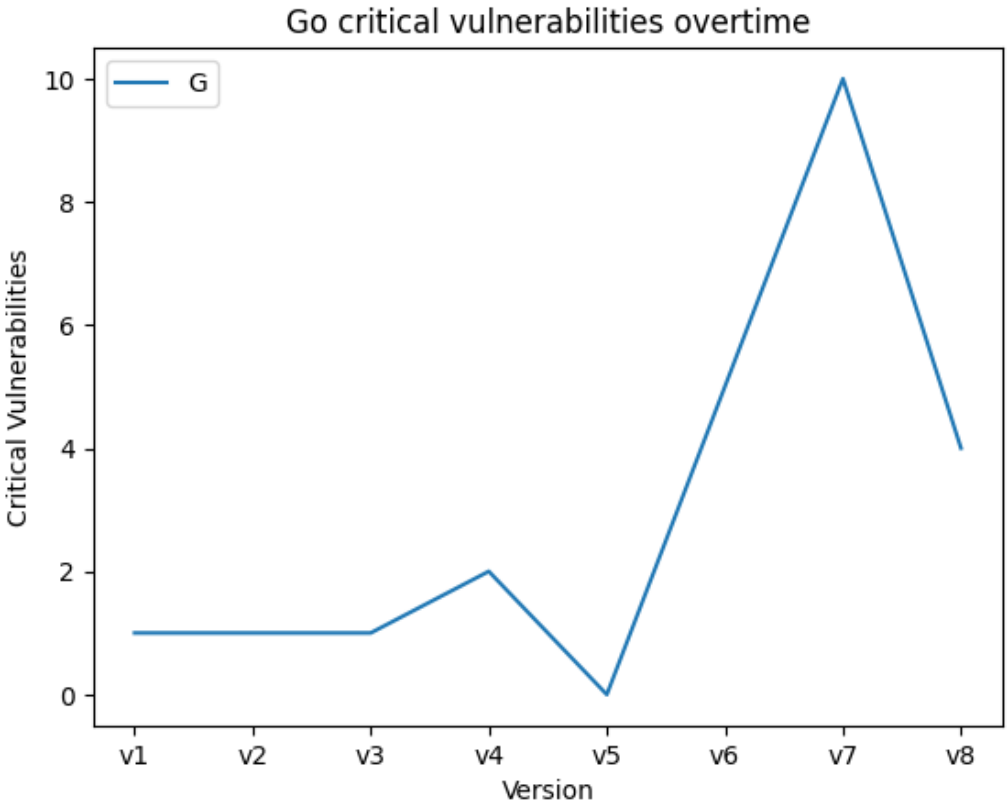


Figure 5.18: GO critical vulnerabilities over time

Project Name	CVSS Vector	CVSS Score
Tomtom	CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H	9.8
kluctl	CVSS:3.1/AV:L/AC:L/PR:N/UI:N/S:C/C:H/I:H/A:H	9.3
Kustomizer	CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:N	9.1
tf-controller	CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:N	9.1

Table 5.10: Top Vulnerabilities Based on CVSS Scores

CWE-ID	Count of Total Occurrences	Presence Across Projects
CWE-78	34	2
CWE-697	6	1
CWE-184	6	1
CWE-639	8	3

Table 5.11: CWE ID Occurrences and Project Presence

5.6 Information about each CWE id's identified

CWE-78, focusing on OS Command Injection, unveiled issues stemming from improper handling of user-controlled input during command execution. The findings indicated a lack of robust validation and sanitation of external inputs, emphasizing the need for stringent input validation and the implementation of parameterized queries. Additionally, adherence to the principle of least privilege was underscored as a pivotal measure to curtail potential damage from command injection attacks.

Moving to CWE-184, which addresses the Incomplete List of Disallowed Inputs, security concerns were associated with inadequate content filtering, potentially leading to bypassing protection mechanisms. The results emphasized the importance of not only identifying disallowed inputs but also maintaining a comprehensive list of allowed inputs. This dual focus ensures a more robust defense against potential threats, with a call for proper encoding of outputs to fortify security measures.

The evaluation of CWE-697, concerning Incorrect Comparison, revealed instances where incorrect checks, missing factors, and inconsistent comparisons occurred, impacting security at various levels. The implications highlighted the necessity for thorough checks, emphasizing the accurate evaluation of security-related elements through meticulous comparisons. Validating multiple factors and utilizing appropriate comparison logic emerged as key practices to eliminate security weaknesses.

Lastly, CWE-639, addressing Authorization Bypass Through User-Controlled

Key, brought to light situations where unauthorized access was possible through the manipulation of keys due to inadequate authorization checks. The findings emphasized the significance of ensuring proper permissions and authorization. Encryption and digital signatures were recommended to bolster data security, preventing external entities from gaining control of critical keys.

Chapter 6

Discussion

This research aims to explore SBOMs which have been an effective way to tackle security challenges in today's software development environment. In this discussion, we will delve deeply into the findings from our research on SBOMs and security integration within Python and Go ecosystems. This exploration not only clarifies the specific outcomes of our study but also places them within the broader context of software security and open-source development.

The Dynamic Evolution of SBOMs (RQ1):

SBOMs are increasing in software development, indicating a focus on security. Analysis of collected SBOM files shows growing complexity in Python and Go ecosystems. Python's SBOMs (that's the list of all the components in a project), had 9 projects with components ranging from 7 to 81 across different versions. It seems like Python's SBOMs are getting more complex, which could mean they have more vulnerabilities. Go's SBOMs, on the other hand, have 6 projects with components ranging from 46 to 1781, and they remain pretty stable. This suggests that Go has different developmental and security dynamics. But, keeping track of all these components can be a huge challenge. Having detailed SBOMs is super important for transparency and security, which are crucial in modern software development. Sometimes, complex things can be a bit tough to handle, especially for small teams or projects with limited resources. This research found that even though there are guidelines for SBOM content quality, it can vary a lot. This means that having guidelines is not enough for all SBOMs to be of the same quality. These differences could cause problems in making sure software is secure, which might lead to missed vulnerabilities. This research also provided a basic outline of what information must be contained in an SBOM file. SBOM has two sections header and package details section. The tables 5.1 5.2 will give a clear idea of what should a quality SBOM should contain. And also weneed more research to find out why there are differences even though there are standards. We looked at the Python and Go ecosystems, so it may not be the same in other programming environments. The research suggests that there is a need for more efficient tools to manage SBOMs. Future studies can explore ways to balance the level of detail and usability of SBOMs and extend these practices to other programming environments. The evolution of SBOMs has significant societal implications, particularly in ensuring the security of increasingly complex software systems. From an ethical standpoint, it raises questions about collective responsibility in maintaining secure open-source software.

Policy Impacts on SBOM Quality (RQ2):

This study looked at how the US Executive Order affected the usage of SBOMs in two ways. First, SBOM files before and after the Executive Order were checked to see if there were any changes in quality. The analysis focused on the impact of the US Executive Order of May 2021 on SBOM quality and revealed significant improvements post-intervention. The Sbomqs tool that was used showed that there was a clear improvement in SBOM quality after the order, which is good for software security practices. The improvement was assessed based on structural elements, NTIA minimum elements, semantic quality, and sharing quality. The quality score of SBOMs improved from 6.475 before the Executive Order to 8.275 after it. This means that not only did they follow NTIA guidelines, but the overall quality of SBOMs also got better. The NTIA minimum elements score went up from 6.400 to 9.225, and the semantic quality score improved from 3.900 to 7.225. All of these numbers show that SBOMs became better and easier to understand post-intervention. Another way to look at it is that some projects follow the rules better than others. After the Executive Order, the quality score standard deviation went down from 0.222 to 1.118. This means that more projects met quality standards, but it also shows that not all projects apply the rules in the same way. The quality of SBOM has been improved with the help of policy mandates that have raised software security standards. We did some analysis using Ordinary Least Squares (OLS) regression and also performed an interrupted time series. We found that the model is pretty good at explaining the quality scores of Software Bill of Materials (SBOMs) with an R-squared value of 0.847, which means 84.7% of the variability in SBOM quality scores can be accounted for by the predictors. We also found that the model is robust and not overfitted with an adjusted R-squared value of 0.785. The F-statistic of 13.80 and a significant p-value show that the predictors we included are significant in explaining SBOM quality. Based on the coefficients, the intercept (const) represents the quality score when both Time and Time-Period are zero, and it is estimated to be 7.675. The coefficient for Time shows that on average, the quality scores decrease by 0.480 for each additional unit of time, suggesting a decline in SBOM quality over time. However, the coefficient for Time-Period is positive (3.720) and statistically significant, indicating a significant increase in the quality score after the intervention. This finding aligns with the US Executive Order of May 2021 has a positive impact on improving SBOM quality. While this policy intervention has led to an overall improvement in compliance with NTIA guidelines and better SBOM semantic understandability, it has been observed that the application of these policies across different projects has been uneven. The study found that policy interventions can improve the quality of software security practices. However, flexible policies are needed to accommodate the diverse nature of open-source projects. Improving SBOM quality has broader societal implications, enhancing the security and transparency of essential software. It would be interesting to explore how different policies affect software ecosystems and develop more flexible rules. The quality of SBOMs has improved after the policy intervention, which is great for enhancing the transparency and security of software that is essential to modern infrastructure. This shows that both developers and policymakers are taking responsibility to ensure that software supply chains are trustworthy. The US Executive Order on SBOMs has positively influenced SBOM quality, but the way it's applied varies, so we need

more adaptable and detailed policy frameworks for open-source software. This thesis examined eight different projects before and after May 2021. Initially, the aim was to evaluate the quality of SBOMs by analyzing historical git data from projects that had SBOMs before and after the intervention (May 2021). However, we were unable to find repositories that had SBOMs in both timelines, which is a limitation of this research. If repositories are discovered that have SBOMs in both timelines, future research can compare them more effectively.

Security Scans Integration and its Efficiency (RQ3):

This investigation focused on the combination of security scans with SBOMs resulting in various insights. On one hand, these scans have proven to be crucial in managing vulnerabilities, particularly in revealing how critical vulnerabilities evolve. For example, Tomtom, a Python project, accounted for about 75 percent of all identified critical vulnerabilities, indicating significant code quality and security issues. In contrast, Macron had significantly fewer critical vulnerabilities, with only 31, and several projects like SBOMnix, lerentis-bitwarden-crd-operator, cli-plugin, and cve-bin-tool reported no critical vulnerabilities at all. This difference in security risks shows that some projects are riskier than others. It also proves that not all vulnerabilities are the same across all projects. When we dug deep into the vulnerabilities and found some interesting stuff. We found 34 instances of OS Command Injection vulnerabilities (CWE-78) across two projects. We also found six instances each of Incorrect Comparison (CWE-697) and Incomplete List of Disallowed Inputs (CWE-184) in one project each. Plus, we identified eight instances of Authorization Bypass Through User-Controlled Key (CWE-639) across three projects. It just shows how diverse and complex the vulnerabilities can be in open-source software. We examined 16 open-source software projects, focusing on the most critical vulnerabilities. The vulnerabilities were evaluated based on their CVSS Scores (ranging from 9.1 to 9.8) and CWE IDs. It was observed that Tomtom and Macron had over 90 percent of the critical vulnerabilities, indicating high severity. Python projects had more critical vulnerabilities than Go projects, with an average of 14.9 critical vulnerabilities per version in Python and only 3.6 in Go. This highlights the need for different security strategies for different programming environments. The gap between the number of critical vulnerabilities in Python and Go projects seems to be reducing over time, suggesting a changing landscape in software security. However, it's important to understand the limitations of this research. It is important to note that the selection of projects from mainly Python and Go ecosystems may not fully represent the diversity of open-source software. Moreover, depending on certain vulnerability scanning tools could lead to biases due to their varying sensitivities and detection capabilities. The diverse distribution of vulnerabilities suggests that future research should focus on developing more strong security strategies that address specific project vulnerabilities. Additionally, by exploring the reasons behind the absence of critical vulnerabilities in certain projects, we may gain valuable insights into effective security practices. The fact that widely used software has critical weaknesses has serious consequences for society, which means that we need to take security seriously when it comes to digital infrastructure. We also need to think about who's responsible for making sure that open-source software is as safe as possible. To identify vulnerabilities, security scans that are integrated with SBOMs play a crucial role. But we also

need to keep in mind that vulnerabilities are not evenly spread out across different projects. This means that we need to keep improving our security practices and strategies, and maybe even come up with different ways to deal with vulnerabilities depending on the situation.

Comparative Ecosystem Analysis: Python vs. Go:

Comparative analysis of the Python and Go ecosystems revealed distinct trends in vulnerability across various projects, such as Tomtom and SBOMnix. The number of vulnerabilities in Python projects ranged from 7 to 81 across different versions, with a general increasing trend over time. For instance, SBOMnix had 34 vulnerabilities in version 0.2.0 but was free of vulnerabilities in version 0.6.9. On the other hand, Go projects like Bom initially had an increase in vulnerabilities, but it was followed by a significant decrease. The Go projects had a wider range of components, from 46 to 1781, and showed more stability with some gradual increases. The Go project flux started with 55 vulnerabilities, which was reduced to 4 in later versions, showing a 90 percent reduction, and effective response to security risks.

It's important to note that the analysis only focused on certain projects within each ecosystem, so it may not reflect the larger dynamics. Additionally, the results could be affected by the particular techniques and tools used to identify vulnerabilities. Therefore, it's important to acknowledge these limitations to get a more balanced understanding of the findings, which highlights the need for more comprehensive research and a variety of methods. Moreover, Python projects had an average of 48.8 components, while Go projects had an average of 420.5, reflecting different scales of project complexity. Python's incremental evolution in SBOMs suggests a gradual identification and mitigation of vulnerabilities, whereas Go updates brought more volatile vulnerability shifts. This difference underscores the unique security challenges of each ecosystem and points to the necessity of ecosystem-specific security strategies. In summary, a comparative analysis of Python and Go ecosystems revealed distinct vulnerability trends and security challenges. This highlights the importance of adapting security strategies to address evolving software vulnerabilities. Future research could focus on understanding the factors contributing to the observed trends in vulnerabilities and developing ecosystem-specific security strategies. Additionally, exploring the reasons behind the consistent vulnerabilities in some Python projects and the initial high vulnerabilities in Go projects would provide valuable insights.

Chapter 7

Threats to Validity

7.1 Internal Validity

It is a measure of the extent to which a study's results can be attributed to the independent variable and not to other factors [42]. To address threats to internal validity, several key measures were implemented in the research design. One of the primary strategies employed was the use of a randomized control trial (RCT) design [43]. This approach involved the random assignment of software projects to either the control group, which did not receive the SBOM implementation, or the experimental group, which underwent SBOM implementation. Random assignment was crucial as it helped to ensure that any observed differences in outcomes could be confidently connected to the approach - in this case, the SBOM implementation.

Additionally, the inclusion of a control group helped mitigate potential threats to internal validity. The control group, which did not undergo the SBOM intervention, served as a baseline for comparison. By having a group that remained unaffected by the intervention, the study minimized the potential influence of extraneous variables that could have otherwise confounded the results. This approach significantly strengthened the internal validity of the research.

Furthermore, to establish causality and further address internal validity threats, a pre-post testing methodology was adopted. This involved the measurement of SBOM quality metrics and vulnerability assessments both before and after the SBOM implementation within each group. By conducting these assessments at two different time points, the study aimed to track changes over time within each group. This approach allowed for a thorough examination of the impact of the SBOM implementation on software security, reinforcing the internal validity of the research design.

7.2 External Validity

External validity refers to the extent to which the findings of a study can be generalized to other populations, settings, and times. In other words, external validity is a measure of whether the results of a study can be applied to real-world situations [44]. To make sure that the research results can be applied to a wide range of software projects and that there are no external validity threats. A diverse sample of projects from the Python and Go ecosystems were chosen. This way, the study's findings will be more broadly applicable to a variety of open-source projects and not just a

narrow range of them.

The study was done in real-world software development practices so that the research's findings would be relevant and useful in actual scenarios. By studying real software projects, the research aimed to discover the complexities and differences that are usually encountered in practical software development. This approach helped to address external validity threats by grounding the study in the reality of software development practices and making the findings more useful to real-world situations.

7.3 Construct Validity

Construct validity refers to the extent to which a test measures what it is intended to measure. In other words, construct validity is a measure of whether the test is assessing the underlying construct or trait that it is designed to measure [45]. During the research process, we were really careful to make sure we were measuring everything accurately so that our results would be trustworthy. We were especially worried about two things: SBOM quality metrics and vulnerability assessments. To help us out, we came up with really clear definitions for these concepts. By doing this, we made sure that we were measuring what we wanted to measure, and that our results were legit.

Additionally, the study used a triangulation method to gather data from various sources such as SBOM files, vulnerability scans, and quality metrics. This approach helped to increase the accuracy of the research by verifying the results through different data sources and methods. Using multiple data sources not only made the study's conclusions stronger but also made it more confident that it was legit.

Chapter 8

Conclusions and Future Work

8.1 Conclusion

As a part of this thesis, the objective was to explore the information present in SBOM files along with their corresponding vulnerability result files. This study aimed to analyze the evolution of SBOM files (RQ1), evaluate the impact of the US Executive Order 2021 on SBOM quality (RQ2), and assess the effectiveness of integrating security scans with SBOMs (RQ3). This section reflects on the findings of this thesis and discusses their implications for the field of software security and SBOMs, addressing several key concerns:

Information in SBOM Files (RQ1): The archival study involved collecting and analyzing SBOM files from Python and Go ecosystems. The findings in Tables 5.1 and 5,2 provide a comprehensive overview of the elements needed to create a quality SBOM. Additionally, Table 5.3 illustrates the information that vulnerability scans give, which helps organizations in enhancing their security practices.

Evolution of SBOMs (RQ1.1): This research provides an in-depth analysis of the evolving role of SBOMs in tracking software components, licenses, and vulnerabilities in the software development life cycle. The study found that Python projects showed a consistent increase in the number of components over time, while Go projects had more variability in their growth. The study also identified significant differences in licenses between Python and Go projects. Python projects tend to have a more permissive license structure due to its compatibility with various dependencies and libraries. On the other hand, Go projects tend to have more restrictive licenses, which reflects their concern for compatibility issues. Vulnerabilities have been reduced over time (latest versions) in both ecosystems. The findings of this research underscore the importance of using SBOMs in tracking software components, licenses, and vulnerabilities to ensure software quality and security in the long run.

Impact of US Executive Order (RQ2): It was found that the US Executive Order had a positive impact on the quality of SBOMs among open-source projects. The average quality score of SBOMs increased significantly from 6.475 to 8.275 after the order was implemented. The NTIA minimum elements score and the semantic quality score also showed significant improvements, indicating the impact of the policy on software development teams.

Integration of Security Scans with SBOMs (RQ3): The integration of security scans with SBOMs revealed a wide range of vulnerabilities. For instance, some projects like Tomtom exhibited a high concentration of critical vulnerabilities, while others showed none. This highlights the need for personalized security approaches to tackle each project's specific vulnerabilities. The use of CVSS scores, ranging from 9.1 to 9.8, along with CWE IDs such as CWE-78 and CWE-697, provided deeper insights into the severity and nature of these vulnerabilities.

To summarize, this thesis highlights how SBOMs are evolving in software security, the positive impact of policy on the quality of SBOMs, and the crucial role of security scans in identifying and mitigating vulnerabilities. These findings underscore the need for adaptive and responsive security practices that are tailored to the unique challenges of different software ecosystems.

8.2 Future Work

In this section, we discuss the future work for this thesis.

- While this study focused on Python and Go ecosystems, future research could extend to other programming languages and ecosystems to provide a more comprehensive understanding of SBOM practices across diverse software development environments.
- The US Executive Order's impact on SBOM quality suggests significant policy implications. Longitudinal studies could be conducted to assess the long-term effects of such policies on software security standards and practices across industries.
- Investigating how end-users interact with SBOMs and their comprehension of these documents could provide insights into how SBOMs can be made more user-friendly and effective in communicating security information.
- Exploring the integration of SBOMs within the DevOps framework could provide valuable insights into how SBOMs can be effectively used in continuous integration/continuous deployment (CI/CD) pipelines for enhancing software security in real-time.
- Researching the role of SBOMs in emerging technologies such as IoT, cloud computing, and blockchain could guide how SBOM practices can be adapted to these new domains.

References

[1] A. Fuggetta, "Open source software—an evaluation," *Journal of Systems and Software*, vol. 66, no. 1, pp. 77–90, 2003.

[2] M. J. Hossain Faruk, M. Tasnim, H. Shahriar, M. Valero, A. Rahman, and F. Wu, "Investigating novel approaches to defend software supply chain attacks," in *2022 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pp. 283–288, 2022.

[3] D. L. Vu, I. Pashchenko, F. Massacci, H. Plate, and A. Sabetta, "Towards using source code repositories to identify software supply chain attacks," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, (New York, NY, USA), p. 2093–2095, Association for Computing Machinery, 2020.

[4] Siteadmin, "UAParser.js npm Package Supply Chain Attack: Impact and Response," 12 2023.

[5] "Log4Shell vulnerability highlights software supply chain issues," 1 2022.

[6] M. Campbell, "Keep your dependencies secure and up-to-date with github and dependabot," May 2021.

[7] M. Ohm, H. Plate, A. Sykosch, and M. Meier, "Backstabber's knife collection: A review of open source software supply chain attacks," in *Detection of Intrusions and Malware, and Vulnerability Assessment* (C. Maurice, L. Bilge, G. Stringhini, and N. Neves, eds.), (Cham), pp. 23–43, Springer International Publishing, 2020.

[8] X. Ding, F. Zhao, L. Yan, and X. Shao, "The method of building sbom based on enterprise big data," in *2019 3rd International Conference on Electronic Information Technology and Computer Engineering (EITCE)*, pp. 1224–1228, 2019.

[9] É. Ó. Muirí, "Framing software component transparency: Establishing a common software bill of material (sbom)," *NTIA, Nov*, vol. 12, 2019.

[10] W. House, "Executive Order on Improving the Nation8217;s Cybersecurity," 5 2021.

[11] M. Crosignani, M. Macchiavelli, and A. F. Silva, "Pirates without borders: The propagation of cyberattacks through firms' supply chains," *Journal of Financial Economics*, vol. 147, no. 2, pp. 432–448, 2023.

[12] CrowdStrike.com, "Cybersecurity 101: Supply chain attacks," Sept. 2023. Accessed on March 8, 2024.

[13] P. Nguyen, S. Durlauf, and M. Tikalsky, *Software Bill of Materials: A Catalyst to a More Secure Software Supply Chain*. PhD thesis, Acquisition Research Program, 2023.

[14] N. Zahan, E. Lin, M. Tamanna, W. Enck, and L. Williams, "Software bills of materials are required. are we there yet?," *IEEE Security & Privacy*, vol. 21, no. 2, pp. 82–88, 2023.

[15] Z. Wei, "Research on the application of open source software in digital library," *Procedia Engineering*, vol. 15, pp. 1662–1667, 12 2011.

[16] "Github." GitHub.

[17] Kinsta®, "What is github?," Nov. 2023. Accessed on March 8, 2024.

[18] F. Perez, B. E. Granger, and J. D. Hunter, "Python: an ecosystem for scientific computing," *Computing in Science & Engineering*, vol. 13, no. 2, pp. 13–21, 2010.

[19] R. Costanza, R. De Groot, L. Braat, I. Kubiszewski, L. Fioramonti, P. Sutton, S. Farber, and M. Grasso, "Twenty years of ecosystem services: how far have we come and how far do we still need to go?," *Ecosystem services*, vol. 28, pp. 1–16, 2017.

[20] SecurityMetrics, "Vulnerability scanning 101." SecurityMetrics, July 2023. Accessed on March 8, 2024.

[21] Google, "Google/osv-scanner: Vulnerability scanner written in go which uses the data provided by https://osv.dev."

[22] P. Mell, K. Scarfone, and S. Romanosky, "Common vulnerability scoring system," *IEEE Security & Privacy*, vol. 4, no. 6, pp. 85–89, 2006.

[23] B. Martin, M. Brown, A. Paller, D. Kirby, and S. Christey, "Cwe," *SANS top*, vol. 25, 2011.

[24] T. W. House, "Executive order on improving the nation's cybersecurity," 2021.

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

[26] T. Stalnaker, N. Wintersgill, O. Chaparro, M. Di Penta, D. M. German, and D. Poshyvanyk, "Boms away! inside the minds of stakeholders: A comprehensive study of bills of materials for software systems," *arXiv preprint arXiv:2309.12206*, 2023.

[27] T. Bi, B. Xia, Z. Xing, Q. Lu, and L. Zhu, "On the way to sboms: Investigating design issues and solutions in practice," 2023.

[28] D. Buttner, R. A. Martin, and M. C. B. MA, "The cybersecurity benefits of leveraging a software bill of materials," 2022.

[29] A. Chaora, N. Ensmenger, and L. J. 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)*, pp. 1–4, 2023.

[30] P. Caven and L. Camp, "Towards a more secure ecosystem: Implications for cybersecurity labels and sboms," *SSRN Electronic Journal*, 01 2023.

[31] H. Plate, S. E. Ponta, and A. Sabetta, "Impact assessment for vulnerabilities in open-source software libraries," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 411–420, 2015.

[32] B. Gokkaya, L. Aniello, and B. Halak, "Software supply chain: review of attacks, risk assessment strategies and security controls," *arXiv preprint arXiv:2305.14157*, 2023.

[33] J. A. Harer, L. Y. Kim, R. L. Russell, O. Ozdemir, L. R. Kosta, A. Rangamani, L. H. Hamilton, G. I. Centeno, J. R. Key, P. M. Ellingwood, *et al.*, "Automated software vulnerability detection with machine learning," *arXiv preprint arXiv:1803.04497*, 2018.

[34] S. L. Eggers, T. B. Simon, B. R. Morgan, E. S. Bauer, and D. Christensen, "Towards software bill of materials in the nuclear industry,"

[35] N. Zahan, S. Shohan, D. Harris, and L. Williams, "Do software security practices yield fewer vulnerabilities?," in *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pp. 292–303, 2023.

[36] K. Bronstad, "References to archival materials in scholarly history monographs," *RBM: A Journal of Rare Books, Manuscripts, and Cultural Heritage*, vol. 19, no. 1, p. 28, 2018.

[37] T. D. Cook, D. T. Campbell, and W. Shadish, *Experimental and quasi-experimental designs for generalized causal inference*, vol. 1195. Houghton Mifflin Boston, MA, 2002.

[38] J. DiNardo, *Natural Experiments and Quasi-Natural Experiments*, pp. 1–12. London: Palgrave Macmillan UK, 2016.

[39] J. Ferron and G. Rendina-Gobioff, *Interrupted Time Series Design*. John Wiley Sons, Ltd, 2005.

[40] D. McDowall, R. McCleary, and B. J. Bartos, *Interrupted time series analysis*. Oxford University Press, 2019.

[41] S. L. Turner, A. B. Forbes, A. Karahalios, M. Taljaard, and J. E. McKenzie, "Evaluation of statistical methods used in the analysis of interrupted time series studies: a simulation study," *BMC medical research methodology*, vol. 21, no. 1, pp. 1–18, 2021.

[42] K. Cahit, "Internal validity: A must in research designs," *Educational Research and Reviews*, vol. 10, no. 2, pp. 111–118, 2015.

[43] K. Stanley, "Design of randomized controlled trials," *Circulation*, vol. 115, no. 9, pp. 1164–1169, 2007.

[44] L. Baldwin, "Internal and external validity and threats to validity," in *Research concepts for the practitioner of educational leadership*, pp. 31–36, Brill, 2018.

[45] S. M. Downing and T. M. Haladyna, "Validity and its threats," *Assessment in health professions education*, vol. 1, pp. 21–56, 2009.

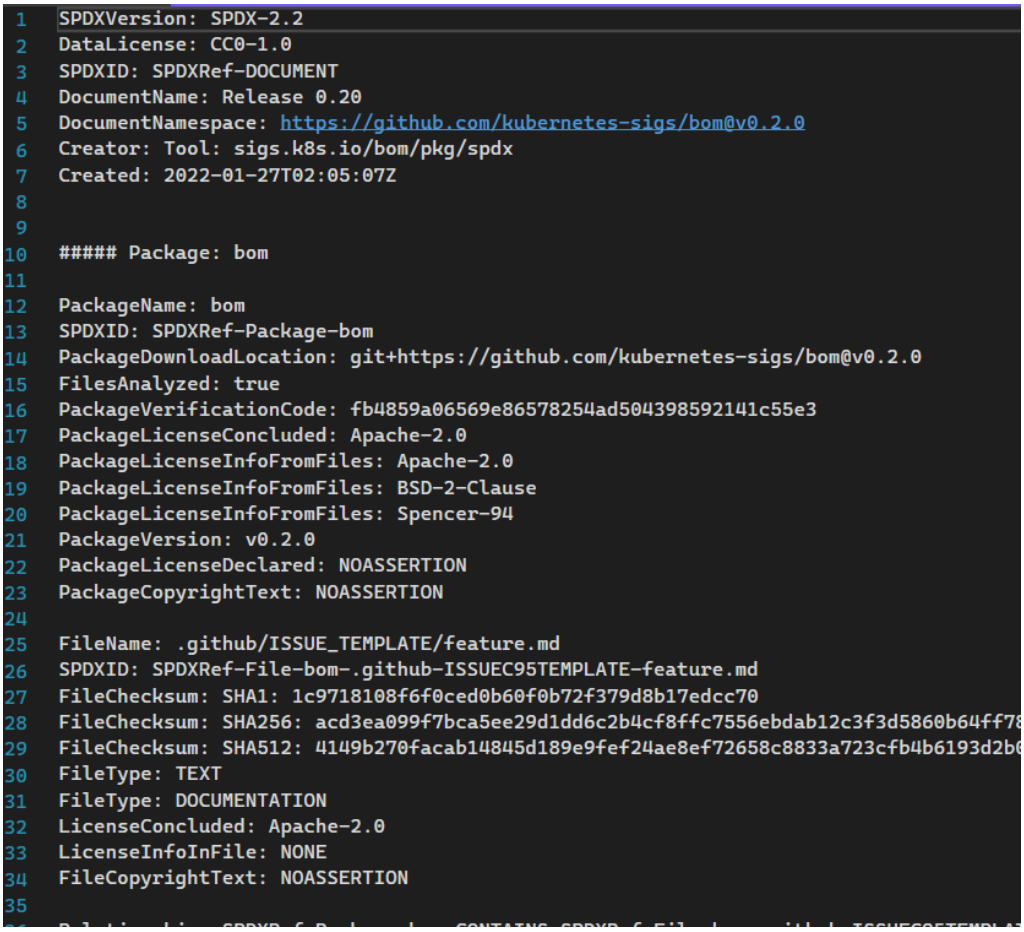
Appendix A

Supplemental Information

article array hyperref

Table A.1: Repository/Project and Link to the Repository

Repository/Project	Link to the Repository
kluctl	https://github.com/kluctl/kluctl
anchore/chronicle	https://github.com/anchore/chronicle/releases
Kustomizer	https://github.com/stefanprodan/kustomizer/releases?page=1
Bom	https://github.com/kubernetes-sigs/bom/releases
Cli-plugin	https://github.com/docker/SBOM-cli-plugin/releases
Tf-controller	https://github.com/weaveworks/tf-controller
flux	https://github.com/fluxcd/flux2/releases
Tomtom-international	https://github.com/tomtom-international/vault-assessment-prometheus-exporter
cve-bin-tool	https://github.com/intel/cve-bin-tool/tree/main/SBOM
Internet.nl	https://github.com/internetstandards/Internet.nl
Lerentis/bitwarden-crd-operator	https://github.com/Lerentis/bitwarden-crd-operator/releases
macaron	https://github.com/oracle/macaron
PyStacks	https://github.com/KablamoOSS/PyStacks/blob/master/pystacks.spdx
SBOMnix	https://github.com/tiiuae/SBOMnix
FOSSology-REST-API	https://github.com/Shruti3004/FOSSology-REST-API/blob/master/src/spdx2/agent/spdx2
boyter/Ic	https://github.com/boyter/Ic



```
SPDXVersion: SPDX-2.2
DataLicense: CC0-1.0
SPDXID: SPDXRef-DOCUMENT
DocumentName: Release 0.20
DocumentNamespace: https://github.com/kubernetes-sigs/bom@v0.2.0
Creator: Tool: sigs.k8s.io/bom/pkg/spdx
Created: 2022-01-27T02:05:07Z


##### Package: bom

PackageName: bom
SPDXID: SPDXRef-Package-bom
PackageDownloadLocation: git+https://github.com/kubernetes-sigs/bom@v0.2.0
FilesAnalyzed: true
PackageVerificationCode: fb4859a06569e86578254ad504398592141c55e3
PackageLicenseConcluded: Apache-2.0
PackageLicenseInfoFromFiles: Apache-2.0
PackageLicenseInfoFromFiles: BSD-2-Clause
PackageLicenseInfoFromFiles: Spencer-94
PackageVersion: v0.2.0
PackageLicenseDeclared: NOASSERTION
PackageCopyrightText: NOASSERTION

FileName: .github/ISSUE_TEMPLATE/feature.md
SPDXID: SPDXRef-File-bom-.github-ISSUEC95TEMPLATE-feature.md
FileChecksum: SHA1: 1c9718108f6f0ced0b60f0b72f379d8b17edcc70
FileChecksum: SHA256: acd3ea099f7bca5ee29d1dd6c2b4cf8ffc7556ebdab12c3f3d5860b64ff78
FileChecksum: SHA512: 4149b270facab14845d189e9fef24ae8ef72658c8833a723cfb4b6193d2b6
FileType: TEXT
FileType: DOCUMENTATION
LicenseConcluded: Apache-2.0
LicenseInfoInFile: NONE
FileCopyrightText: NOASSERTION

```

Figure A.1: Snippet of SBOM file

Figure A.2: Snippet of vulnerability result file

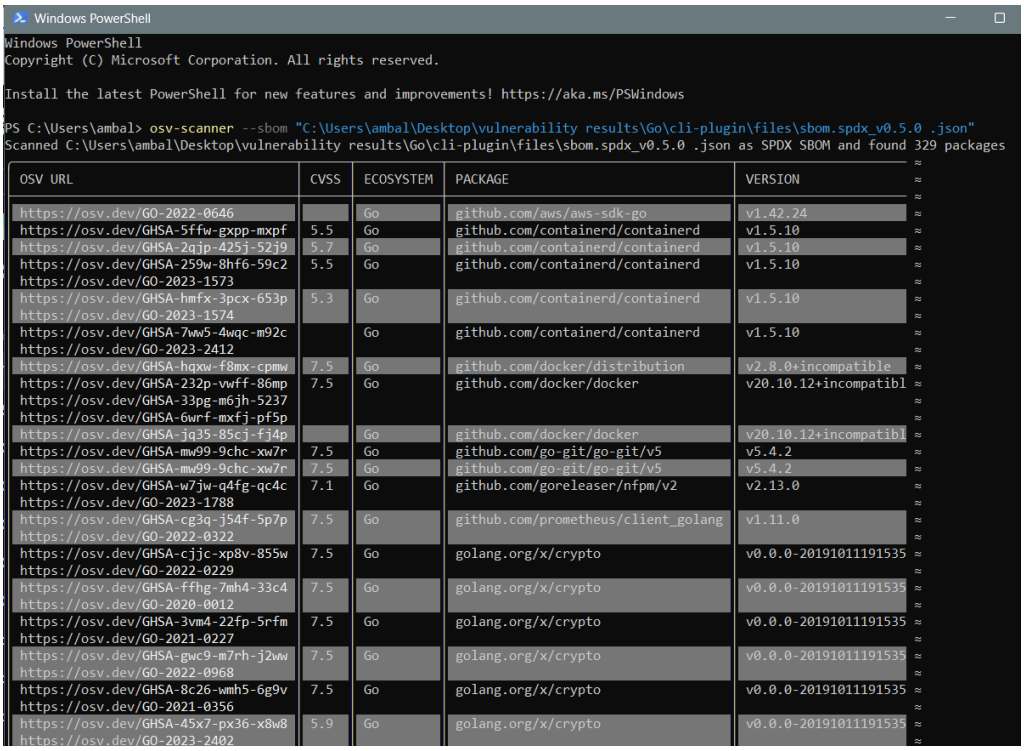
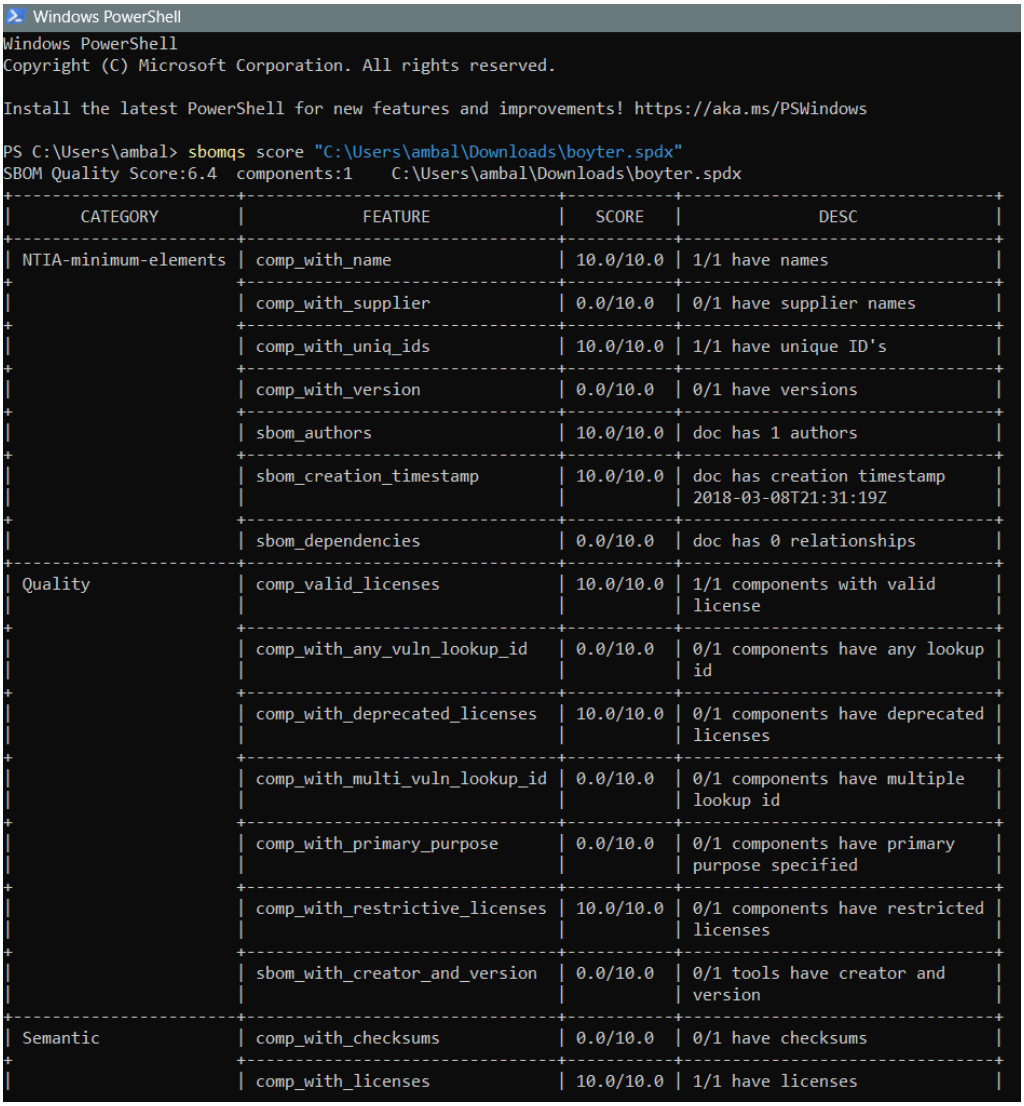


Figure A.3: Example vulnerability scan result using OSV-scanner

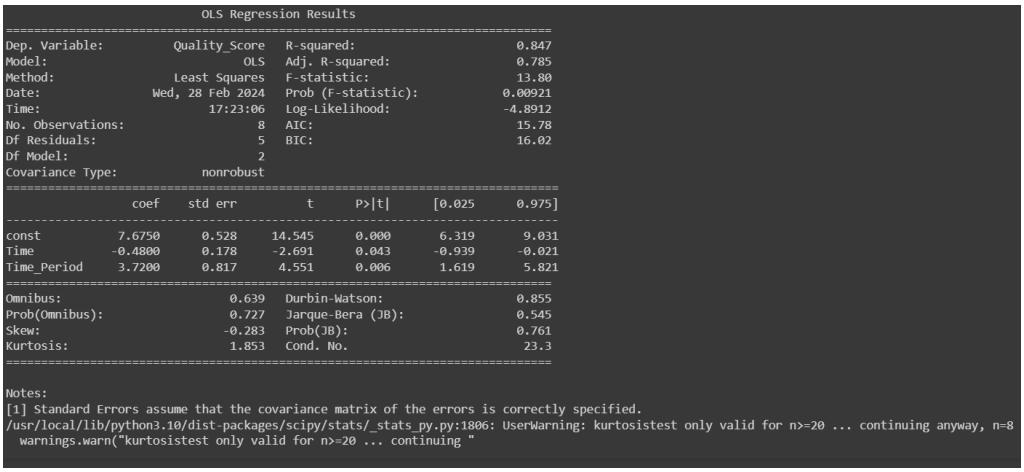


```
Windows PowerShell
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\ambal> sbomqs score "C:\Users\ambal\Downloads\boyter.spdx"
SBOM Quality Score:6.4  components:1    C:\Users\ambal\Downloads\boyter.spdx
+------------------------+---------------------------------+-----------+--------------------------------+
|       CATEGORY         |            FEATURE              |   SCORE   |              DESC              |
+------------------------+---------------------------------+-----------+--------------------------------+
| NTIA-minimum-elements  | comp_with_name                  | 10.0/10.0 | 1/1 have names                 |
+                        +---------------------------------+-----------+--------------------------------+
|                        | comp_with_supplier              | 0.0/10.0  | 0/1 have supplier names        |
+                        +---------------------------------+-----------+--------------------------------+
|                        | comp_with_uniq_ids              | 10.0/10.0 | 1/1 have unique ID's           |
+                        +---------------------------------+-----------+--------------------------------+
|                        | comp_with_version               | 0.0/10.0  | 0/1 have versions              |
+                        +---------------------------------+-----------+--------------------------------+
|                        | sbom_authors                    | 10.0/10.0 | doc has 1 authors              |
+                        +---------------------------------+-----------+--------------------------------+
|                        | sbom_creation_timestamp         | 10.0/10.0 | doc has creation timestamp     |
|                        |                                 |           | 2018-03-08T21:31:19Z           |
+                        +---------------------------------+-----------+--------------------------------+
|                        | sbom_dependencies               | 0.0/10.0  | doc has 0 relationships        |
+------------------------+---------------------------------+-----------+--------------------------------+
| Quality                | comp_valid_licenses             | 10.0/10.0 | 1/1 components with valid      |
|                        |                                 |           | license                        |
+                        +---------------------------------+-----------+--------------------------------+
|                        | comp_with_any_vuln_lookup_id    | 0.0/10.0  | 0/1 components have any lookup |
|                        |                                 |           | id                             |
+                        +---------------------------------+-----------+--------------------------------+
|                        | comp_with_deprecated_licenses   | 10.0/10.0 | 0/1 components have deprecated |
|                        |                                 |           | licenses                       |
+                        +---------------------------------+-----------+--------------------------------+
|                        | comp_with_multi_vuln_lookup_id  | 0.0/10.0  | 0/1 components have multiple   |
|                        |                                 |           | lookup id                      |
+                        +---------------------------------+-----------+--------------------------------+
|                        | comp_with_primary_purpose       | 0.0/10.0  | 0/1 components have primary    |
|                        |                                 |           | purpose specified              |
+                        +---------------------------------+-----------+--------------------------------+
|                        | comp_with_restrictive_licenses  | 10.0/10.0 | 0/1 components have restricted |
|                        |                                 |           | licenses                       |
+                        +---------------------------------+-----------+--------------------------------+
|                        | sbom_with_creator_and_version   | 0.0/10.0  | 0/1 tools have creator and     |
|                        |                                 |           | version                        |
+------------------------+---------------------------------+-----------+--------------------------------+
| Semantic               | comp_with_checksums             | 0.0/10.0  | 0/1 have checksums             |
+                        +---------------------------------+-----------+--------------------------------+
|                        | comp_with_licenses              | 10.0/10.0 | 1/1 have licenses              |
```

Figure A.4: Example quality score result using SBOMqs

```python
import statsmodels.api as sm
import pandas as pd

# Create a DataFrame with the provided data
data = {
    'Project': ['Fossology', 'Internetstandards', 'Pystacks', 'Boyter', 'cve-bin-tool', 'Lerintiz', 'bom', 'Kluctl'],
    'Quality_Score': [6.8, 6.3, 6.4, 6.4, 9.4, 8.8, 8.1, 6.8],
    'Time_Period': [0, 0, 0, 0, 1, 1, 1, 1]  # Numerical coding: 0 for 'Before', 1 for 'After'
}

df = pd.DataFrame(data)

# Add a time variable based on the index (assuming time order)
df['Time'] = range(1, len(df) + 1)

# Create a design matrix manually
X = sm.add_constant(df[['Time', 'Time_Period']])

# Fit the regression model
model = sm.OLS(df['Quality_Score'], X)
result = model.fit()
print(result.summary())
```

Figure A.5: Code used for Interrupted time series analysis



```
                            OLS Regression Results
==============================================================================
Dep. Variable:          Quality_Score   R-squared:                       0.847
Model:                            OLS   Adj. R-squared:                  0.785
Method:                 Least Squares   F-statistic:                     13.80
Date:                Wed, 28 Feb 2024   Prob (F-statistic):            0.00921
Time:                        17:23:06   Log-Likelihood:                -4.8912
No. Observations:                   8   AIC:                             15.78
Df Residuals:                       5   BIC:                             16.02
Df Model:                           2
Covariance Type:            nonrobust
================================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
--------------------------------------------------------------------------------
const          7.6750      0.528     14.545      0.000       6.319       9.031
Time          -0.4800      0.178     -2.691      0.043      -0.939      -0.021
Time_Period    3.7200      0.817      4.551      0.006       1.619       5.821
==============================================================================
Omnibus:                        0.639   Durbin-Watson:                   0.855
Prob(Omnibus):                  0.727   Jarque-Bera (JB):                0.545
Skew:                          -0.283   Prob(JB):                        0.761
Kurtosis:                       1.853   Cond. No.                         23.3
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
/usr/local/lib/python3.10/dist-packages/scipy/stats/_stats_py.py:1806: UserWarning: kurtosistest only valid for n>=20 ... continuing anyway, n=8
  warnings.warn("kurtosistest only valid for n>=20 ... continuing "
```

Figure A.6: Result of Interrupted time series analysis

Faculty of Faculty, Blekinge Institute of Technology, 371 79 Karlskrona, Sweden