Tracking Risk Across Time: A Multi-Format SBOM Model for Vulnerability Evolution in Software Projects

Vidit Gala[252IS012]*, Rahul Kumar[252IS029]*, Akshay Kumar*
*Department of Computer Science and Engineering
*National Institute of Technology Karnataka
*Surathkal Mangalore - 575025, India
*Email - [viditgala.252is012; rahul.252is029; al.akshay.cs]@nitk.edu.in

***Abstract*—This paper discusses an SBOM-driven software supply chain risk management paradigm that maps a software project across versions for vulnerabilities. Using SBOMs generated in CycloneDX and SPDX formats, enriched with information from OSV and NVD, the model creates longitudinal vulnerability timelines and dependency-change (dependency-churn) profiles. The model utilizes semantic version sorting, computes dependency churn, and leverages a map-reduce–based aggregation method to identify high-consequence security trends across versions of software. We examined the model on some Python and Java repositories and observed the model accurately produces a list of newly added and removed vulnerabilities, changes in severity, and dependency risks across software releases (i.e., commits). We performed experimentation/scalability tests on ten larger sized projects, demonstrating significant performance increases from parallelization and caching as we analyzed as the size of the repo grew substantially decreased exploitation time while retaining a defensible analytical return on investment. Results validate that the model produced actionable investigation into changes in security posture, with recommended safer versions over time.**

***Index Terms*—SBOM, CycloneDX; SPDX; Vulnerability Mapping; OSV; Dependency Churn; Longitudinal Analysis.**

I. Introduction

The software supply chain has become one of the most important areas in cybersecurity due to the rapid growth of open source ecosystems, cloud-native architectures and dependency-rich development workflows. Modern software applications are no longer monolithic and consist of hundreds of interconnected libraries, transitive dependencies, container layers, APIs and third-party services. This means that a single compromise in any upstream component can be affected downstream to thousands of products, making supply chain attacks appeal to adversaries due to their potential impacts. High-profile incidents such as SolarWinds, Log4Shell, Codecov and dependency-confusion attacks highlight that attackers exploit indirect entry points (e.g.: build pipelines, package registries, CI/CD systems and outdated components) to introduce the attacks instead of targeting software vendors directly. The environment has prompted global standards bodies like NIST, CISA, OWASP and the new European Union Cyber Resilience Act, to encourage visibility, transparency and traceability throughout the entire software lifecycle. In addition, within this domain, Software Bills of Materials (SBOMs) have emerged as foundational artefacts to enumerate every component in products so that organizations are able to analyze security posture, track vulnerable dependencies or quickly respond to threats. As the software ecosystem continues to grow in complexity, the demand for automated, scalable, and longitudinal methods for analyzing software supply chain risks has become indispensable, making SBOM-driven analysis a central area of research and operational security practice.

There are benefits to SBOMs including improved identification of vulnerabilities, increased license compliance, and support for security updates in large-scale distributed systems. Pursuant to SBOM adoption in practice, domains such as healthcare, IoT/IIoT, finance, and defense are increasingly reliant on SBOM-based monitoring as a risk assessment tool within their respective industries. However, there are real-world obstacles to SBOM adoption they are often (1) incomplete, (2) inconsistent, (3) lacking standardization, and (4) manual installation packages often omit pertinent information. Previous studies tell us that SBOMs are low in their adoption within open source projects, and that many artifacts across releases are transient or non-persistent. In addition, we see that most assessments show static snapshots, wherein there is no longitudinal view of how vulnerabilities change over time.

Prior research has explored the topic of SBOM usage, effectiveness of SBOM tools, blockchain-assisted verification, and AI-based risk assessment of SBOMs. For example, Kawaguchi et al. showed that frequently utilized SBOM tools fail to account for packages manually installed that were not specified in the Docker image causing gaps in oversight. Sabato et al. indicated that SBOMs created for open-source repositories often contain not only large gaps, but are also not standardized. Other studies looked at formal verification for secure firmware updates, or frameworks for C-SCRM, but most were based in a controlled experiment without the ability to find vulnerabilities in real-world applications, or from the level of a git commit. These studies indicate the need for a practical, scalable, multi-format SBOM analysis framework that can track vulnerabilities per historical version, not single

versioning.

The paper fills these gaps by proposing a practice-research hybrid, empirical, SBOM-driven vulnerability-analysis framework that. It generates SBOMs for each version, augment these SBOMs with existing vulnerability information from OSV and NVD and constructs security timelines. Key contributions include:

- Longitudinal analysis of vulnerability evolution by commit across multiple releases.
- Comparative analysis of dual-format SBOM (CycloneDX & SPDX) for evaluating consistency and completeness.
- Automated dependency-churn assessment using semantic sorting and set-difference logic.
- A scalable enrichment pipeline, facilitated by parallelization and caching, tested on up to ten large projects.
- Actionable security insights (e.g., identifying safer versions, CVSS trends, and chained high-risk dependencies).

II. Related Works

In this section, we provide an overview of the related work for 10 journal papers. Topics include cyber supply chain risk management (C-SCRM) and SBOM adoption, evaluation and enhancement. Subtopics include C-SCRM frameworks aligned with NIST/RMF, AI-driven risk and conformity assessment. The SBOM-focused works analyze adoption trends, quality issues and tooling performance (precision, recall, efficiency), explore license compliance and propose secure update mechanisms for IIoT devices.

Authors JUNG et al. [1] created and validated a Cyber Supply Chain Risk Management (C-SCRM) framework specifically for the military by reviewing major standards like NIST SP 800-161, RMF and ENISA. They identified 32 key information types and synthesized supply chain threat categories, which were then used to develop a six-phase framework (Prepare, Collect/Share, Define, Assess, Respond, Monitor) that aligns with military procurement and RMF processes. The framework's practicality was confirmed through a case study involving open-source software and practitioner feedback, though the authors noted that existing frameworks often lack certain information types and the process still requires manual risk measurement, standardized controls and further validation.

Sabato et al. [2] conducted an empirical study on the adoption of Software Bill of Materials (SBOM) in open-source projects by mining GitHub for files and tools, resulting in a curated dataset of 186 projects using SPDX or CycloneDX. Their analysis focused on SBOM storage, completeness and persistence in the codebase or release artifacts. The findings indicated that while SBOM adoption is low, it is increasing; however, many generated SBOMs are incomplete, nonstandard and often not retained for downstream use. The study's limitations include the potential for false positives or negatives in their mining methodology, the temporal bias of their data, limited understanding of project motivations for adoption and the high-level nature of their quality checks.

Bamidele Samuel Adelusi et al. [3] proposed a blockchain-integrated SBOM framework to enable real-time vulnerability detection in decentralized package repositories. Their system combines SBOM generation, blockchain storage and smart contract-based verification with CVE/NVD threat intelligence for automated alerts. The framework, tested on over 15,000 components from PyPI and npm, demonstrated near real-time detection and high accuracy (F1 score of 0.94, 96.2% precision and 91.8% recall), with good scalability and acceptable blockchain transaction overhead. The authors noted that the evaluation was based on simulations and permissioned blockchain assumptions, emphasizing that the accuracy of the system is contingent on the freshness of the SBOM data.

Kitty Kioskli et al. [4] developed an AI-driven Risk and Conformity Assessment (RCA) framework to enhance the security of healthcare systems and medical supply chains. Their multi-layered model used various machine learning techniques, including LSTM and CNN for anomaly detec- tion, Bayesian networks for risk scoring and reinforcement learning for optimizing firewall and patch management. The prototype, tested in a hospital network, processed device logs, telemetry and vulnerability data. Evaluation results were promising and resulted in 97% accurate detection, 78% reduction in firewall misconfigurations and a 62% reduction in vulnerability exposure. The framework also showed impovments in risk assessment accuracy by 23% compared to traditional manual methods. The authors, however, noted that the framework's performance relies on high-quality input data, its integration is complex and further validation across multiple hospitals is necessary.

Alessia Antelmi et al. [5] analyzed FOSS license usage patterns using the Software Heritage Analytics framework on 835 popular GitHub repositories across eight languages. They applied ScanCode to detect declared and in-code licenses, identified multi-class licensing and conflicts and used statistical tests to examine relationships with project size and language. Results showed that only 25% of projects were single-licensed, with large projects having many license pairs and frequent conflicts, especially in C and OS-related projects. The authors noted that focusing on top-starred repositories may bias results and that the analysis covers only the latest project revision, not historical license evolution.

Eman Abu Ishgair et al. [7] performed a qualitative evaluation of software supply chain security techniques by mapping defense mechanisms to specific security objectives and comparing them against 72 known attacks from the IQT Labs dataset. Their **ASTRA** model was validated through case studies of the SolarWinds and Left-pad software supply chains, demonstrating how various defenses can address particular threats and enhance security. However, the authors acknowledged several limitations, including the lack of human-centered usability studies, the absence of a quantitative cost-benefit analysis for each defense mechanism and the inherent performance and efficiency trade-offs associated with ledger-based systems.

Nobutaka Kawaguchi et al. [8] analyzed 3,514 Docker container images from Docker Hub to evaluate the performance of SBOM generation tools. They specifically focused on images

with manually installed packages to identify those with known vulnerabilities. Their findings showed that SBOM tools often missed these manually installed or actively used packages and that the extracted version information was usually inaccurate. The study's limitations include its focus only on Docker Hub, its "snapshot in time" nature and the inability to accurately measure recall due to a lack of a definitive "ground truth" for the installed packages and the failure to account for false positives generated by the SBOM tools.

Tingting Bi et al. [9] studied SBOM design issues by analyzing 4,786 discussions from 510 GitHub projects using Grounded Theory. They annotated the discussions to identify common challenges related to traceability, dependencies, licensing, tooling and non-functional requirements. The study found that the average resolution time for SBOM issues was 20.3 days and highlighted challenges such as technical debt and tooling gaps, noting the potential role of AI and machine learning. The authors acknowledged limitations, including the study's restriction to GitHub, potential missed discussions due to keyword searches and possible bias from manual annotation. They concluded that integrating SBOMs with development tools and AI could enhance their adoption and effectiveness.

Alberto Tacchella et al. [10] proposed a framework for securing firmware updates in industrial IoT devices (IIoT) by integrating SBOMs and behavioral certification manifests (BCM) with formal verification. Their method involved extending SUIT manifests, generating proofs using the cvc5 SMT solver and analyzing control flow graphs with angr. They tested their approach on sample manifests, SBOMs and trusted applications from OpTEE OS, finding that manifest parsing scaled linearly, while proof verification time grew quadratically for larger control flow graphs. The authors noted that large proofs could not be verified on low-end devices, sequential verification increased total time and their analysis was limited to memory partitioning and control flow graph invariance properties. Despite these limitations, the study successfully demonstrated the feasibility of formally verified secure updates for IIoT and provided a roadmap for future work on expanding property coverage and optimizing the verification process.

Menghan Wu et al. [11] evaluated the effectiveness of SBOM tools for Java projects in their work "More Than Meets the Eye: On Evaluating SBOM Tools In Java". The authors selected six popular open-source SBOM tools through a systematic literature review and examined their performance across 952 Maven projects. They identified three common component import methods—Build Tool Import (BTI), Dynamic Loading (DL) and Source Code Import (SCI)—and built a benchmark of 152 projects (132 real-world and 30 synthetic) to evaluate the tools in different contexts. The study measured effectiveness (precision, recall, F1-score), time efficiency and detection capability under the three import scenarios. Results showed clear trade-offs between speed and accuracy: for example, Syft was the fastest (5.09s) but had poor accuracy, while Cdxgen was the slowest (315.19s) yet achieved the best dependency detection. Most tools performed well with BTI but struggled significantly with DL and SCI. SbomTool achieved the highest F1-score (92.39%) for component detection, though dependency detection remained weak across tools. The authors noted limitations, including reliance on Maven projects only, restricted ground-truth construction using `.gitmodules` and potential dataset bias due to the focus on popular GitHub projects.

The table I summarizes a comparison of ten relevant studies on SBOM adoption, supply chain risk, vulnerability discovery, license analysis, and tool evaluation. Each study is assessed according to criteria, methodology, findings, and limitations; one of the many themes is the pursuit for greater transparency, accuracy, and automation in software supply chain security. These studies mention limitations including a narrow dataset, snapshot versus longitudinal assessment, reliance on tools, the lack of scalability, and domain-specific constraints which point to a lack of continuity in assessment using longitudinal methods. In conclusion, this table presents the research gap your proposed model aims to address using empirical, multi-version SBOM to support vulnerability analysis.

The common limitations across these 10 related works are limited scope, scalability, dataset constraints, methodology constraints, simulated environments, high complexity, not realistic solutions and limited real-world validation.
Here are 3 common limitations across SBOM's risk management:-

- Data and methodology constraints: Heavy dependence on specific tools (e.g, ScanCode, SPDX/CycloneDX), heuristics, or repositories introduces bias and false positives/negatives.
- Temporal and scalability issues: Most analyzes capture snapshots in time, lack longitudinal validation and struggle with performance or scalability in real-world, large-scale, or resource-constrained environments.
- Limited scope and generalizability: Many studies rely only on a single domain (e.g, military, healthcare, Maven, GitHub) or small curated datasets, making results less representative.

The major contributions of the paper are:

1) **Historical Commit Analysis:** We did not consider only the last commit but the multiple presupposition commits as well in the github repositories of more starred and frequently used projects. This will allow us to trace the way that the vulnerabilities are changing between the various software versions and make a better picture of software security over time.
2) **Comparative SBOM Analysis:** The model generates and compares SBOMs in SPDX and CycloneDX format in an attempt to ascertain the strengths and weaknesses of each one compared to vulnerability detection and the model of dependencies.
3) **Real-world Project Implementation:** The methodology is applied to genuine open-source projects hosted on GitHub, ensuring that the analysis reflects real-world software dependencies and build environments.

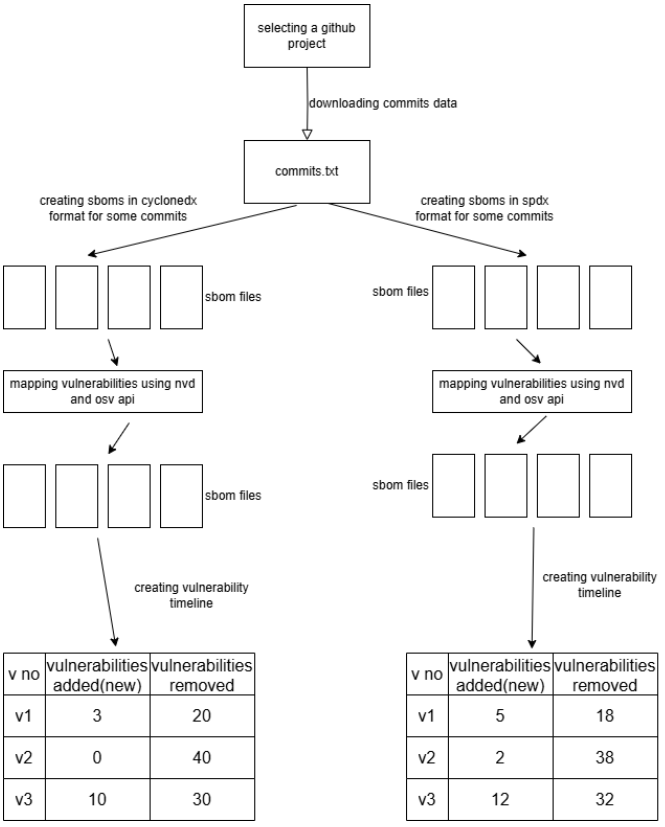
TABLE I
SUMMARY TABLE OF STUDIES

Sr. No.	Author	Metrics	Methodology	Results	Limitations
[1]	Jung, Cho et al. (2025)	Coverage metric, mapping, Case-study applicability, Practitioner feedback	Reviewed major frameworks (NIST SP 800-161,etc) → derived 32 information types → designed 6-phase C-SCRM framework → validated via OSS onboarding case study	Identified 6 missing information types in US frameworks; proposed domain-specific solution; improved risk visibility.	Manual risk measurement, Limited scope, Conceptual work, No standardization of controls, Model only for military
[2]	Sabato et al. (2025)	Adoption count, Availability metric, Standards compliance, Temporal trend indicators, Recommended-info coverage	Mined GitHub for SBOM files (SPDX/CycloneDX), manually curated 186 projects, checked SBOM completeness and storage practices.	SBOM adoption low but growing, many SBOMs incomplete, non-standard, or transient.	Sample size, False positives and negatives, Temporal bias, show "what" but not always the "why", SBOM quality measures are coarse
[3]	Adelusi et al. (2023)	Detection Latency, Detection Accuracy, Throughput, Resource Efficiency, F1 score, Recall	Built blockchain-based SBOM system using Hyperledger Fabric, smart contracts, CVE feed integration and CI/CD automation; simulated decentralized repo environment	Demonstrated near real-time detection and high accuracy (F1 score of 0.94, 96.2% precision and 91.8% recall), with good scalability	Evaluation in simulated environment, No address for 0 day vulnerabilities, SBOM freshness depends on CI/CD integration, Performance results were depended on a blockchain
[4]	Kioskli et al. (2025)	F1 score, True Positive and False Positive Rates, Risk Assessment, Reduction in Misconfigurations, Patch Management Efficiency	Designed AI-driven risk assessment + conformity model using LSTM, CNN, Bayesian networks, etc and deployed the prototype in hospital network.	F1 - 0.92, TPR - 91%, FPR - 3%, 97% anomaly detection accuracy; 78% fewer firewall misconfigs, 62% reduced exposure time	Dependence on high-quality training data, Limited Scope, Implementation complexity is high requiring ML expertise, High computational cost and latency
[5]	Antelmi et al. (2024)	Number of multi-class license pairs, Number of unique licenses per project, Number of license conflicts	Analyzed FOSS license on 835 popular GitHub repositories then applied ScanCode to detect multiple type of licenses and conflicts	Results showed that only 25% of projects were single-licensed, with large projects having many license pairs and frequent conflicts	Results may not generalize for other use cases, Limited Dataset, The analysis relied on ScanCode's detection database, Results may differ on commodity systems
[7]	Abu Ishgair et al. (2024)	Mapping of security objectives to defense techniques; coverage of prior attacks	ASTRA model (Artifacts, Steps, Resources, Principals, Topology) as DAG to represent software supply chain; applied to identify critical security objectives	Identified objectives for each ASTRA element (e.g., P1–P5 for Principals, A1–A3 for Artifacts); no single defense sufficient for full risk mitigation	Excludes human-related threats, lacks usability focus and underexplores real-world deployment cost and adoption challenges
[8]	Kawa - guchi et al. (2024)	Overlooked packages, active usage, packages associated with known CVEs; performance differences of SBOM generators	Empirical analysis of 3,514 Docker images; Generative AI extracted package metadata and monitored runtime activity via eBPF	51% of containers contained manually installed packages; tools overlooked 30–70%; at least 1.1% of packages both overlooked, actively used and associated with CVEs	Single-repository dataset, tool version bias, partial runtime analysis and manual validation reduce result coverage and accuracy
[9]	Bi et al. (2024)	Discussion frequency, issue resolution time, unresolved issue rate	Repository mining and Grounded Theory analysis of 4,786 GitHub discussions across 510 projects	Identified four SBOM life cycle phases; three main issue categories,65.3% of all observed SBOM issues remain unresolved; 11 development activities highlighted	GitHub dataset limited to SBOM, search term limitations and platform bias impact generalization
[10]	Tacchella et al. (2025)	Feasibility and scalability on IIoT benchmarks; parsing, proof verification, memory usage	SUIT framework extended with SBOM and Behavioral Certification Manifest (BCM); formal proofs encoded as SMT problems for automated verification	The approach ensures secure updates with SBOM and BCM but requires remote verification because prototype testing revealed that proof verification time scales quadratically	Local verification infeasible for constrained IIoT devices,High verification cost and no adaptive threat modeling limit real-world deployment.
[11]	Menghan Wu et al. (2025)	Precision, Recall, F1-score, Time Efficiency, dependency detection across BTI, DL, SCI scenarios	Comparative empirical study of six SBOM tools on 152 Java projects; ground truth included BTI, DL and SCI scenarios	SBOM tools' component detection varies (F1: 37%-92%), dependency detection is weak (max F1: 54.57%), speed-accuracy trade-offs exist.	Maven build projects only for dataset, limited import checks and static analysis reduce generalization and detection accuracy

III. Proposed Methodology

In this section, we discuss in detail about the approach to proposed model, datasets used, the terminologies, the algorithms used and the flow of the model proposed.

A. Overview



v no	vulnerabilities added(new)	vulnerabilities removed
v1	3	20
v2	0	40
v3	10	30

v no	vulnerabilities added(new)	vulnerabilities removed
v1	5	18
v2	2	38
v3	12	32

Fig. 1. Flowchart of proposed model

*1) **Flowchart**:* The proposed model aims to analyze the evolution of software vulnerabilities across the different project versions by using Software Bill of Materials (SBOMs) and vulnerability mapping through trusted databases. Figure 1 presents the complete flow of the proposed model. The flowchart depicts a process for analyzing vulnerabilities in a GitHub project. First, a project is selected and its commit data is downloaded into a file named commits.txt. From these commits, SBOMs are created in two formats — CycloneDX and SPDX. Each set of SBOM files is then analyzed using the NVD and OSV APIs to identify known vulnerabilities. The identified vulnerabilities are mapped for different commits, and a vulnerability timeline is created for each format. These timelines track how many vulnerabilities were newly added or removed in each version. The final output shows comparative tables for both CycloneDX and SPDX formats, illustrating the evolution of vulnerabilities across project versions. Additionally, dependency changes such as additions and removals are captured to highlight potential shifts in security exposure. This end-to-end workflow ensures a comprehensive, commit-wise understanding of how a projects risk posture develops over time.

2) Datasets: For this study, information was generic data beneffited from a publicly available open-source project from GitHub. The project selected was for a Python or Java project that has a -nearly shown-high number of stars to confirm interest and activity from the community. The reason to select this specific project is because it is widely used as a library dependency in different software projects it represents a valid and realistic example to do SBOM (Software Bill of Materials) analysis. Information on the repository was used from the commit history to obtain multiple versions of dependency data to compare, and find additions and removals of dependency or change of those dependencies, and the difference over time. This commit-level data was essential for the dataset used for comparison, aggregation, and trend analysis. Table II shows the how the dataset is selected

```
                                        sha           author  \
0  70298332899f25826e35e42f8d83425124f755a5     Nate Prewitt
1  6e4134b204f675268296b2b44c2d52c8a7927b2c  dependabot[bot]
2  420d16bc7ef326f7b65f90e4644adc0f6a0e1d44     Nate Prewitt
3  3c8decb92fb9062cac4bdd12d6c0d32fb3a2d119  dependabot[bot]
4  5a270e2a95cdf84e746f26face11efdda3aa0127     Nate Prewitt

                                        email                  date  \
0                          nate.prewitt@gmail.com  2025-10-15T11:45:42Z
1  49699333+dependabot[bot]@users.noreply.github.com  2025-10-13T16:16:05Z
2                          nate.prewitt@gmail.com  2025-09-09T09:00:19Z
3  49699333+dependabot[bot]@users.noreply.github.com  2025-09-09T00:09:47Z
4                          nate.prewitt@gmail.com  2025-09-09T00:08:49Z

                                        message
0  Merge pull request #7042 from psf/dependabot/g...
1  Bump github/codeql-action from 3.30.0 to 4.30....
2  Merge pull request #7026 from psf/dependabot/g...
3  Bump actions/setup-python from 5.6.0 to 6.0.0 ...
4  Merge pull request #7025 from psf/dependabot/g...
```

Fig. 2. Dataset

The figure 2 displays a dataset of a github project containing five attributes related to software commits: sha (a unique identifier), author, email, date and message (the commit description). For the model the choice is apparently chosen to use only the sha as input, likely for tasks of data.

TABLE II
Dataset Selection and Exclusion Criteria

Criterion	Justification
Inclusion Criteria	
Hosted on GitHub	Ensures programmatic access to source code, commit history, and release metadata via the GitHub API.
>1000 Stars	Serves as a measure for the popularity and relevance of a project, as it filters out hobby or inconsequential projects.
>500 Commits	Helps ensure the project is sufficiently long and detailed in development history for meaningful temporal analyses.
Exclusion Criteria	
Inconsistent Build	Filters out projects where automated SBOM generation is unreliable, which would compromise data integrity.

B. System Architecture

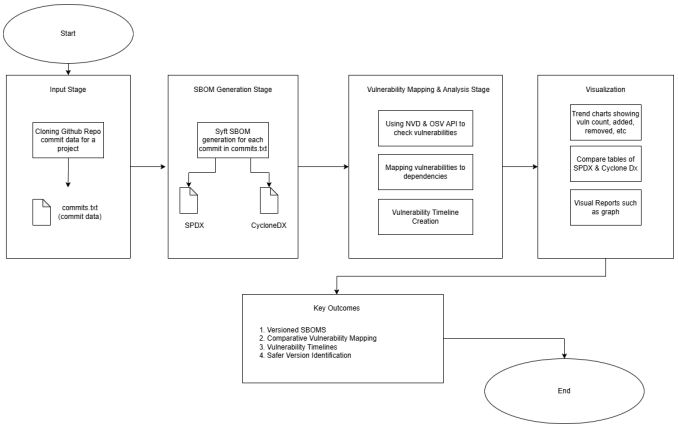


Fig. 3. Architectural Diagram

Figure 3 shows the architecural diagram for the proposed methodology. The SBOM-based Vulnerability Analysis Model streamlines the process of tracking security risks across a project's lifecycle. It starts in the Input Stage by cloning a GitHub repository to gather commit history. The SBOM Generation Stage then utilizes tools like Syft to create comprehensive, versioned SBOMs (in formats like SPDX and CycloneDX) detailing dependencies for every recorded commit. Following this, the Vulnerability Mapping & Analysis Stage maps these SBOM components to known security flaws by consulting vulnerability databases like NVD and OSV, thereby generating crucial vulnerability timelines for each version. Finally, the Visualization Stage presents the findings through reports, comparison tables, and trend charts, with the overall goal being to produce key outcomes such as versioned SBOMs, mapped vulnerabilities, and the identification of future safer dependency versions.

C. Core Methodology Algorithms

*1) **Semantic Version Sorting Algorithm**: -*

To ensure version numbers are compared numerically. This is a custom semantic version sorting algorithm. Converts versions like v1.2.3-rc1 to [1, 2, 3, 1] so they sort numerically rather than lexicographically. This algorithm sorts software package versions based on semantic versioning rules (major.minor.patch). It ensures that dependencies are compared accurately across commits or releases. Figure 4 shows the algorithm flowchart.

Let each version string S = "vX.Y.Z" for eg - (v1.2.10)
f(S) = [x_1, x_2, x_3,...x_n]
where f(S) is a function of vector of integers representing version hierarchy
where x_i = integer component of version string
Eg: f("1.2.10") = [1,2,10]

these all vectors 'S' are then sorted:
Sort(S_1, S_2, S_3, ... S_n) such that it is sorted in ascending manner

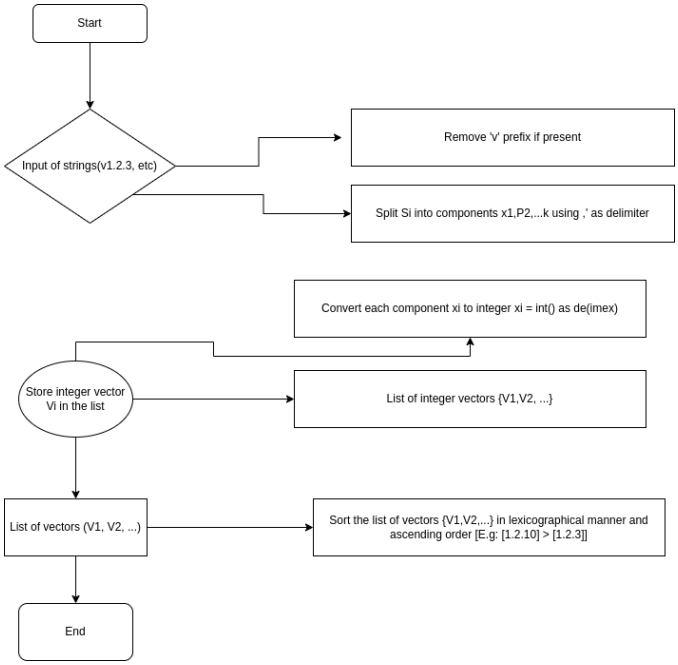


Fig. 4. Semantic sorting algorithm

Function SemanticSort(VersionList):
Input: A list of version strings
Output: A list of versions sorted in semantic order
i. Initialize an empty list TransformationList
ii. For each version string S in VersionList:
a. S_normalized = Replace("v", "", S)
b. S_normalized = Replace("rc", ".", S_normalized) // handle pre-releases
c. S_normalized = Replace("-", ".", S_normalized)
d. Components = Split(S_normalized, ".")
e. Initialize empty list V
f. For each component C in Components: Append Integer(C) to V
g. Append (V, S) to TransformationList // store vector and original string
iii. Sort TransformationList in ascending lexicographic order based on vector V
iv. Extract and return the original version strings (second element) from TransformationList as SortedList
v. Return SortedList

2) Set Difference Algorithm: Efficiently computes added/removed dependencies. Uses mathematical set difference to find newly added and removed dependencies in different versions. The Set Difference algorithm identifies elements present in one set but not in another. It is used to find dependencies that were added or removed between two commits' SBOMs. This helps in analyzing software evolution and identifying potential risk due to dependency changes. Figure 5 shows the algorithm flowchart.

Added dependencies = $D_i \mid D_i - 1$

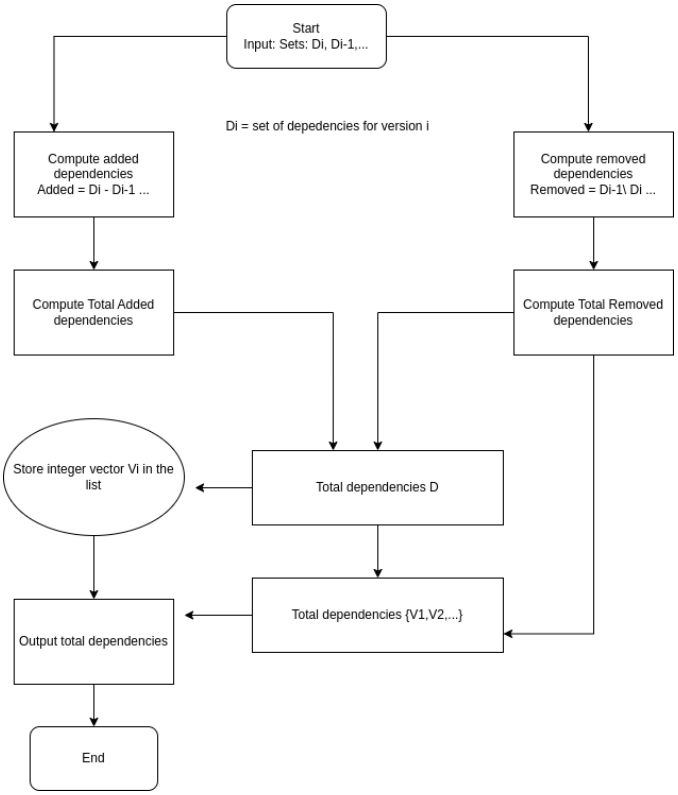


Fig. 5. Set Difference Algorithm

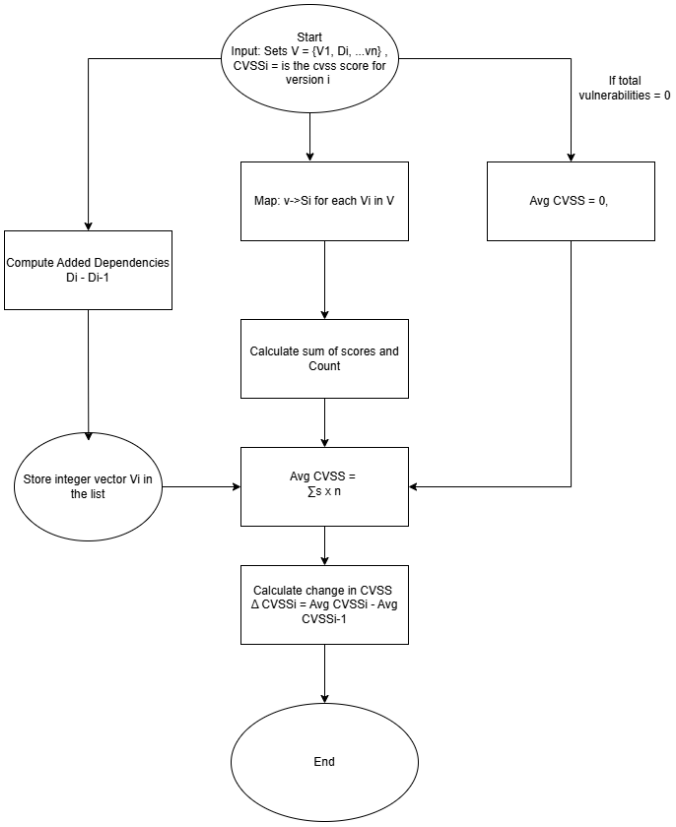


Fig. 6. Aggregation Algorithm

Removed dependencies = $D_i - 1 \mid D_i$

Total Dependencies = D_i

where D_i = set of dependencies in version i, $D_i - 1$ = set of dependencies in version i-1 and $\mid$ represents the set difference between 2 elements

Function ComputeDependencyChurn(DependencySets): Input: DependencySets — a list where each element represents D_i, the set of dependencies for version i

Output: MetricsList — a list of dependency churn metrics for each version

i. Initialize MetricsList = Empty List ii. Initialize PrevDependencies = Empty Set

iii. For i from 1 to Length(DependencySets): a. CurrentDependencies = DependencySets[i]

b. AddedDependencies = CurrentDependencies PrevDependencies

c. RemovedDependencies = PrevDependencies CurrentDependencies

d. TotalDependencies = Size(CurrentDependencies)

e. Metrics = 'Version': i, 'DepsAdded': Size(AddedDependencies), 'DepsRemoved': Size(RemovedDependencies), 'TotalDeps': TotalDependencies

f. Append Metrics to MetricsList

g. PrevDependencies = CurrentDependencies

iv. Return MetricsList

3) Aggregation (Map-Reduce): Sums vulnerability counts, averages CVSS. Similar to a reduce/map operation (data aggregation). The Aggregation algorithm, implemented via a Map-Reduce approach, processes large-scale dependency data by mapping each dependency to key-value pairs and then reducing them to summarize metrics such as total vulnerabilities or average version changes. It is used to efficiently compute overall statistics from multiple SBOMs across different commits. Figure 6 shows the algorithm flowchart.

V = v_1, v_2, v_3,....v_n = set of all vulnerabilities in a version.

each v_i has a CVSS score s_i where CVSS score is a quantitative measure which ranges from 0 to 10 and tells how impactful a vulnerability is, where 0 is low and 10 is critical

TotalVulnerabilities = $|V| = n$

if $n > 0$ AvgCVSS score = $\frac{\sum_{i=1}^{n} s_i}{n}$

if $n = 0$ AvgCVSS score = 0

Change in average CVSS: $\Delta CVSS_i$ = $AvgCVSS_i$ - $AvgCVSS_a$

where a = i-1

$\Delta CVSS_i$ = change in average CVSS score for ith version

Function ComputeReleaseMetrics(D): *Input*: D — list of vulnerabilities for a given release

Output: Dictionary containing total vulnerabilities and average CVSS score

i. Initialize ScoresList = Empty List

ii. For each vulnerability (v) in D: a. Let (s = v)'s CVSS score b. If (s greater than 0), Append (s) to ScoresList

iii. TotalVulnerabilities = Size(D) iv. TotalScores = Size(ScoresList)

v. If TotalScores greater than 0: a. SumScores = Sum(ScoresList) b. AvgCVSS = SumScores / TotalScores Else: c. AvgCVSS = 0

vi. Return 'TotalVulnerabilities': TotalVulnerabilities, 'AvgCVSS': AvgCVSS

D. Proposed Evaluation Methodology

1) Proposed Model: The methodology starts by selecting a GitHub project and cloning it and extracting its commit history. The commits reflect specific states of the software and, through the analysis of its states over time, a single paper leads to a tracking of when vulnerabilities are added or removed. This commits data is then saved in a text file, commits.txt, which is a single input variable for future SBOM generation. Two different SBOM types, CycloneDX and SPDX, will be produced for a fixed set of commits to enable comparative analyses of vulnerability timeline.

SBOMs (Software Bill of Materials) are generated using a SBOM generation tool, *Syft*, for each commit. The SBOM files describe the list of dependencies, libraries, and packages used in the project at that commit in time. SBOMs will be generated in the *CycloneDX* and *SPDX* formats. The SBOM generation tool Syft will automate the extraction of software dependency information from the GitHub project. Initially, all Git tags or commits are enumerated, and each version is checked out in sequence to run two formats to produce an SBOM in CycloneDX JSON and SPDX JSON for that version. Each SBOM is then persisted in a structured output directory for later comparison and analysis.

Upon generating the SBOMs, the model will compare the SBOMs across commits to identify changes in dependencies and associated vulnerability information. The model loads each pair of adjacent SBOMs from both formats in sequence, extracts the dependency names and versions, and then identifies which dependencies were added or removed between each version. It then produces a text summary of the differences for each format. Lastly, it writes these findings to a CSV file that will allow for dependencies to be compared for changes between the CycloneDX and SPDX forms. This analysis is intended to identify differences across the two forms while also identifying dependencies that may represent potential vulnerabilities..

Next each SBOM will be processed to indicate known vulnerabilities using two primary vulnerability database: NVD (National Vulnerability Database) API & OSV (Open Source Vulnerability) API. The model combines both vulnerability APIs to detect security issues in the dependencies found in each SBOM.

2) Pseudo Code:

Script 1: Analyzer

i. Load vulnerability data for all releases.

ii. Compute longitudinal metrics: vuln_count, avg_cvss_score, new_vulns, cvss_change.

iii. Compute dependency churn: deps_added, deps_removed, total_deps.

iv. Merge metrics and churn data → save as `summary.csv`.

Script 2: Git/SBOM Generation

i. For each project in `./projects`:

a. Retrieve all Git tags (versions).

b. For each tag:

1. Checkout the tag version.

2. Generate SBOM using `syft` (CycloneDX JSON).

3. Save SBOM file in `./sboms`.

4. Restore original Git branch.

Script 3: Enrichment & Analysis (Main Pipeline)

i. For each project in `./sboms`:

a. For each SBOM file:

1. Parse dependencies.

2. Query vulnerability source (OSV/NVD).

3. Save enriched data as vulns.json.

Script 4: Visualization

i. Load `summary.csv`.

ii. Generate charts using Matplotlib:

Vulnerability counts vs. releases

Average CVSS score vs. releases

Dependency churn (Added/Removed) vs. releases

iii. Save charts as image files.

E. Methodological Summary

The SBOM-based Vulnerability Assessment model presented is one that can automatically track the software risks across several versions of a project in GitHub repositories. The model relies on workflows to generate SBOMs for each project version using Syft, assesses these against the OSV and NVD databases for software vulnerabilities, and calculates the longitudinal metrics of: vulnerability trends (or volatility) and dependency volatility. Comparison of the CycloneDX and SPDX formats allows for assessment of efficiency and vulnerabilities. Finally, visualization of the results uses summary charts depicting information like vulnerability counts, average CVSS scores, and number of dependencies across releases.

IV. Experimental Result

In this section, we discuss in detail about the experimental setup required for implementation of the project, the evaluation or performance metrics used for checking the results of the model, the results section and finally the scalability analysis.

A. Experimental setup

1) Hardware Components: The designed vulnerability enrichment and analysis pipeline was carried out on a mid-range system that provided a stable environment for SBOM processing and vulnerability identification. The configuration consisted of an **Intel Core i5 processor** with (12 cores), 2.4 GHz, **16 GB DDR4 RAM**, and a **1 TB HDD**, to enable efficient read and write operations during the processing of

SBOM and JSON files. The system's 64-bit architecture, as well as **multithreading**, allowed the dependency parsing, enrichment, and visualization to take place smoothly. The experiments were performed on **Ubuntu 22.04 LTS (64-bit)**, which has been selected for it great compatibility with open-source tools **(Syft and Grype)**. A stable broadband connection was provided throughout the Zoom sessions to fetch real-time vulnerability feeds from the OSV and NVD. Both physical and virtual setups proved to be successful in testing the pipeline.

2) Software Components: In terms of software, the environment consisted of Python 3.10 for data manipulation and parsing, as well as a selection of libraries essential for data extraction and analysis, including **pandas requests** and **json**. The primary software components used were **Syft** for SBOM generation and Grype for vulnerability scanning. Version control and repository management for the project were done using Git and GitHub CLI. The project also took advantage of the **CycloneDX** and **SPDX** Python libraries to validate and transform the SBOM data type. All experiments were performed in a virtual environment (venv), which was used to separate dependencies, allow reproducibility, and ensure consistency across project runs.

B. Performance metrics

Performance metrics were established to assess the outcome of the vulnerability enrichment and analysis procedure. These performance metrics are described as follows:

- *Version Number (Commit Version)*
- *Number of New Vulnerabilities Added (V_{new})*
- *Number of Vulnerabilities Removed ($V_{removed}$)*
- *Average CVSS Score ($CVSS$)*
- *Severity Classification (S)*

The **Version Number** is a time reference, denoting each unique release or commit of the software over time, which allows for assessing dependency security changes between each version.

The **Number of New Vulnerabilities Added** is the numerical increase in vulnerabilities reported in the current version from vulnerabilities present in the prior version:

$$V_{new} = |V_{current} - V_{previous}| \tag{1}$$

Similarly, the **Number of Vulnerabilities Removed** records vulnerabilities fixed or removed between two consecutive versions:

$$V_{removed} = |V_{previous} - V_{current}| \tag{2}$$

The **Average CVSS Score** is a numerical summary of the overall level of risk associated with the vulnerabilities present in a version:

$$CVSS = \frac{1}{N}\sum_{i=1}^{N} CVSS_i \tag{3}$$

where N is the total number of vulnerabilities, and $CVSS_i$ is the score of the i^{th} vulnerability.

Lastly, the **Severity Classification** (S) turns the continuous CVSS score interval from 0 to 10 into a categorical classification scale using standardized thresholds:

$$S = \begin{cases} \text{Low}, & 0.1 \leq CVSS < 4.0 \\ \text{Medium}, & 4.0 \leq CVSS < 7.0 \\ \text{High}, & 7.0 \leq CVSS < 9.0 \\ \text{Critical}, & 9.0 \leq CVSS \leq 10.0 \end{cases} \tag{4}$$

Overall, these performance metrics provide a more comprehensive picture of security over time across versions - assessing the new vulnerabilities added or vulnerability removed, the severity of the vulnerabilities, and monitoring the direction of overall risk of vulnerabilities in software dependencies.

C. Result analysis

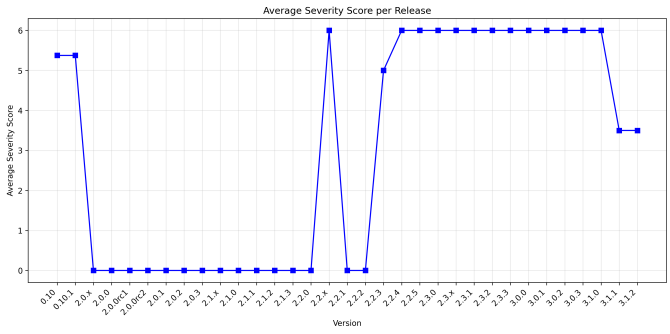


Fig. 7. Average Severity Score per Release

The figure 7 shows the trend of average vulnerability severity scores across Flask versions. In the first few releases (0.x-2.0.x); vulnerability severity was mixed and high-severity scores appeared in various 0.x and 2.0.x releases. As software progressed through the 2.x or later Flasks releases its average severity score reached a plateau in the high range (6.0). This indicates that consistent medium-high level vulnerability issues with medium or high severity were probably an ongoing concern for Flask software development. The score lowered slightly in the last release (3.x) indicating either improved security or patches have been effective. In total; the trend indicates how Flask's vulnerability profile has changed over time as well as some of the higher severity issues in the software development process were either addressed or improved.

The figure 8 the distribution of severity by level of severity in the vulnerability trend data across versions of Flask. High severity vulnerabilities (red line) make up the majority of the vulnerabilities on most releases; there are multiple high severity vulnerabilities in the major releases around version 0.10.x, even more concerning are several high severity vulnerabilities found in the 2.2.x. 2.3.x and 3.0.x releases. There are only weak IO low severity vulnerabilities (green) now and then. Additionally medium and critical issues did not appear in any iterations throughout the vulnerability data. The trends in vulnerabilities indicate that Flask issues meant security issues

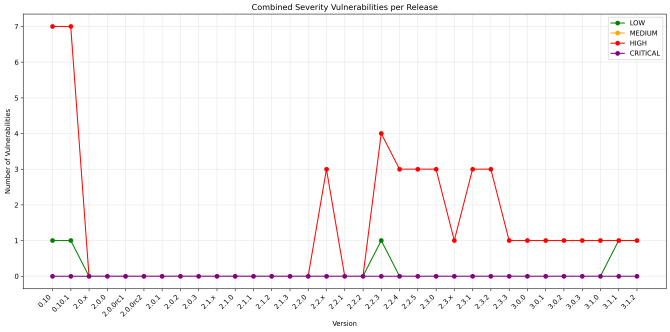


Fig. 8. Combined Severity score trends

historically have largely been high severity vulnerabilities. Indicating a priority for security patches. Finally, the trend of absent critical vulnerabilities suggests that the project has done well avoiding extreme risk over some period of time.

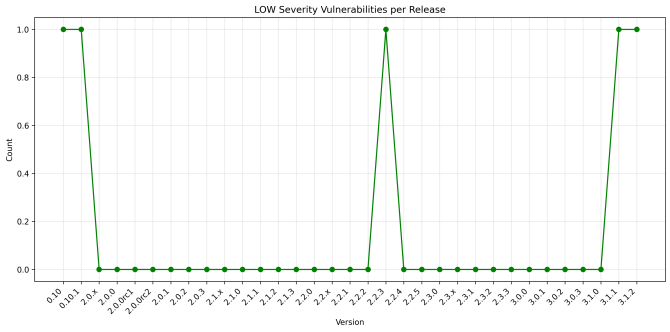


Fig. 9. Low Severity score Trend

The figure 9 indicates that low-severity vulnerabilities are comparatively rare in Flask's lifetime. Only a few releases, especially 0.10.x, 2.2.3, and 3.1.x, recorded a single low-severity vulnerability, and most versions had no vulnerabilities at all. This is a sparse distribution of vulnerabilities that portrays minor issues that detect and resolve quickly, rather than providing a general sense of stability in the code base as a close to stable product. The shortage of low-severity vulnerabilities indicates that Flask developers have had good "code hygiene" and have handled minor risks over time.

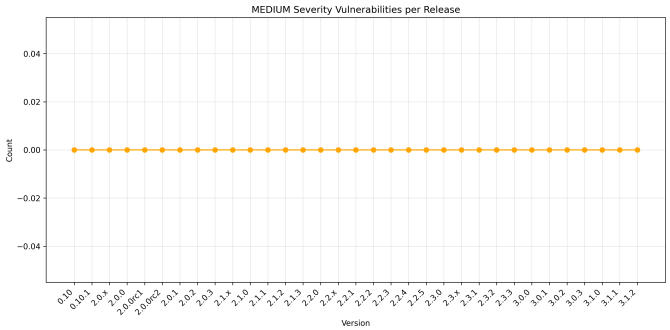


Fig. 10. Medium Severity score Trend

The figure 10 shows data for medium-severity vulnerabilities also remains flat at zero. Medium-level meaningful vulnerabilities were not reported in the analyzed versions. This continued trend indicates regularity in the Flask package in avoiding medium impact vulnerabilities that could affect reliability or user data. The data on medium vulnerabilities adds to the snapshots from the other severity levels to reaffirm that the majority of vulnerability incidents reported on Flask leverage high or low severity and do not often center around medium severity. Overall it highlights the project follows reasonable security practice across versions.

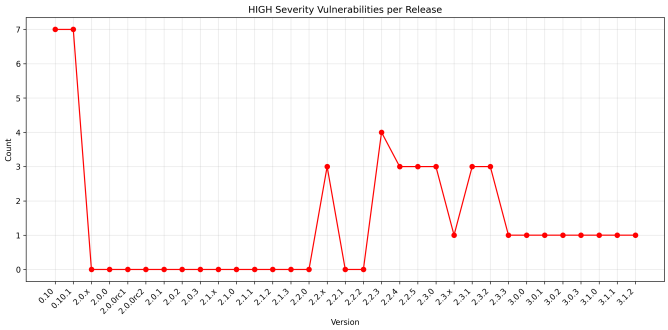


Fig. 11. High Severity score Trend

The figure 11 displays that high-severity vulnerabilities are the most prominent overall in Flask's lifetime. Early versions (0.10.x) had a notable spike with seven high-severity vulnerabilities and a period with several fixes before recurring vulnerabilities between versions 2.2.x to 3.0.x. This dairy occurrence of vulnerabilities suggests that while major vulnerabilities had been found and fixed, some new vulnerabilities occurred in the subsequent releases. The general pattern suggests ongoing high-vulnerability risks, which suggests ongoing security reviews will need to occur during major release upgrades.

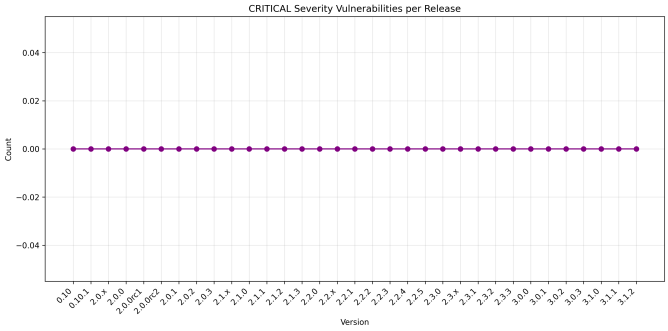


Fig. 12. Critical Severity score Trend

The figure 12 reveals that there were no critical vulnerabilities present in any of the Flask release versions. The horizontal line at the zero line in each version indicates that Flask has consistently provided a strong defense against the most impactful vulnerabilities. The presence of this type of data suggests that Flask has some effective security reviews and testing in place to avoid critical-type vulnerabilities. As a

critical-level vulnerability is often thought of as a catastrophic flaw, it is not surprising to see consistent data of zero indicating a stable and secure code base that assures users they are not prone to these severe types of breach.

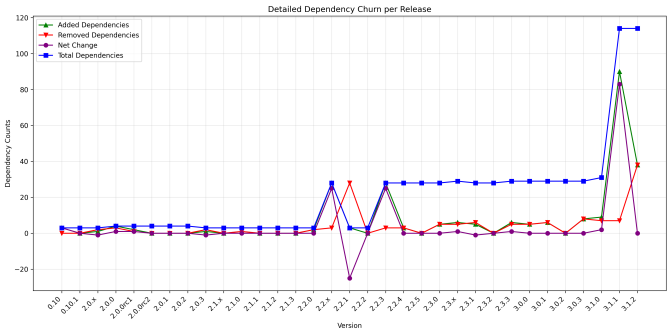


Fig. 13. Deep Churn Trend

The figure 13 illustrates how dependencies changed from one Flask version to the next, including the number of added, removed, and net changes to dependencies. For most versions up to and including 2.1.x, activity around dependencies was limited. A noticeable change occurs around versions 2.2.x and again near 3.1.x. The added dependencies reported in version 2.1.x were further restructured among earlier versions as the underlying framework changed in behavior. By the time to the higher dependency version (3.x) was introduced, the total amount of dependencies rose considerably with a peak of almost, possibly over, 100 unique dependencies in the more recent Flask version, which would indicate new and more functionality and/or integrations. These increases in dependencies may represent larger frameworks that the Flask devs utilized, which is important because it could increase the exposures of security due to new and/or changes in the dependencies.

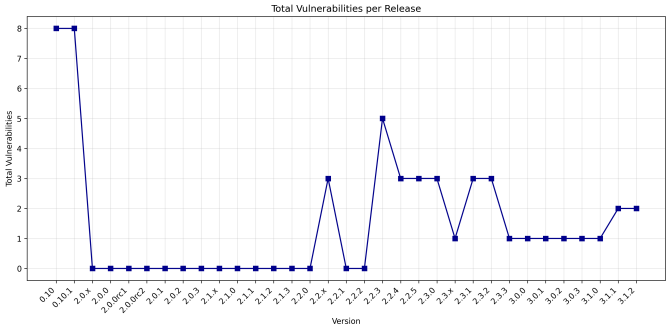
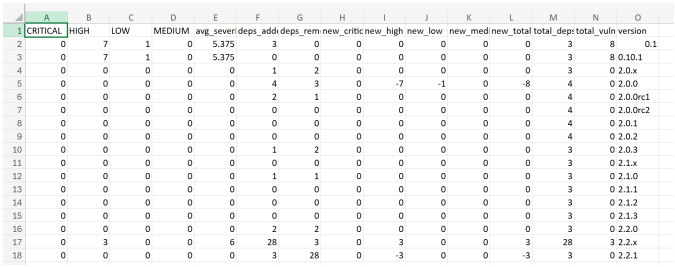


Fig. 14. Total Vulnerability Trend

The figure 14 depicts the total count of vulnerabilities per Flask version. The earliest versions of Flask (0.10.x) had the highest number of recorded vulnerabilities (around 8), with a noticeable decline with later releases. There are again a few spikes in versions 2.2.x through 2.3.x, but these spikes are followed by a leveling out in the 3.x versions that had far fewer vulnerabilities. The declining numbers of vulnerabilities indicates an overall improvement in security practices and managing dependencies over time. The leveling out within the more recent versions indicates a degree of maturity with Flask's development lifecycle and the resolution of the sources of recurring vulnerabilities.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	CRITICAL	HIGH	LOW	MEDIUM	avg_sever	deps_add	deps_rem	new_critic	new_high	new_low	new_medi	new_total	total_dep	total_vuln	version
2	0	7	1	0	5.375	3	0	0	0	0	0	0	3	8	0.1
3	0	7	1	0	5.375	0	0	0	0	0	0	0	3	8	0.10.1
4	0	0	0	0	0	1	2	0	0	0	0	0	3	0	2.0.x
5	0	0	0	0	0	4	3	0	-7	-1	0	-8	4	0	2.0.0
6	0	0	0	0	0	2	1	0	0	0	0	0	4	0	2.0.0rc1
7	0	0	0	0	0	0	0	0	0	0	0	0	4	0	2.0.0rc2
8	0	0	0	0	0	0	0	0	0	0	0	0	4	0	2.0.1
9	0	0	0	0	0	0	0	0	0	0	0	0	4	0	2.0.2
10	0	0	0	0	0	1	2	0	0	0	0	0	3	0	2.0.3
11	0	0	0	0	0	0	0	0	0	0	0	0	3	0	2.1.x
12	0	0	0	0	0	1	1	0	0	0	0	0	3	0	2.1.0
13	0	0	0	0	0	0	0	0	0	0	0	0	3	0	2.1.1
14	0	0	0	0	0	0	0	0	0	0	0	0	3	0	2.1.2
15	0	0	0	0	0	0	0	0	0	0	0	0	3	0	2.1.3
16	0	0	0	0	0	2	2	0	0	0	0	0	3	0	2.2.0
17	0	3	0	0	6	28	3	0	3	0	0	3	28	3	2.2.x
18	0	0	0	0	0	3	28	0	-3	0	0	-3	3	0	2.2.1

Fig. 15. Vulnerability Timeline

The figure 15 provides a comprehensive numerical display of vulnerability assessments and dependencies across several Flask releases. It displays counts for every severity category (Critical, High, Medium, Low) along with average severity scores and dependency churn (added, removed, and total dependencies). For early releases such as 0.10 and 0.10.1, it is noted that higher vulnerability counts exist with most counted as high severity; the later releases show a lower count of vulnerabilities and more stable dependency activity. The zero counts in the critical and medium vulnerabilities remains consistent with the overall range confirming that Flask is resilient against critical dependencies. The dependency metrics suggest that the architecture has added dependencies incrementally over time, especially between 2.2.x, which shows large dependency growth and maintenance.

In summary, the analysis overall indicates that Flask project underwent a brief period of **high-severity vulnerabilities** early on (approximately with version 0.10.x), which were escaped in the later versions. Critical and medium vulnerabilities remained consistent throughout, indicating a strong management of security. There was a spike in dependency churn beginning with version 2.2.x indicating a major update of the framework with slight similarities. Overall, the **results** indicate that Flask continues to **improved** given security stability and dependency in it's evolution.

V. Scalability Analysis

A. Scalability Criteria

The current model evaluates software vulnerabilities by taking two GitHub projects, producing SBOMs for all versions, and mapping vulnerabilities across different releases. To increase scalability and robustness, this analysis has also been extended to additional projects with different complexity and commit histories. This takes the evaluation of vulnerability evolution across different codebases to a broader level. Furthermore, common and broadly used libraries like **Pandas** and **NumPy** were included in the analysis so that the assessment of vulnerabilities would represent real-world high-impact dependencies. The inclusion of an increased complexity with popular

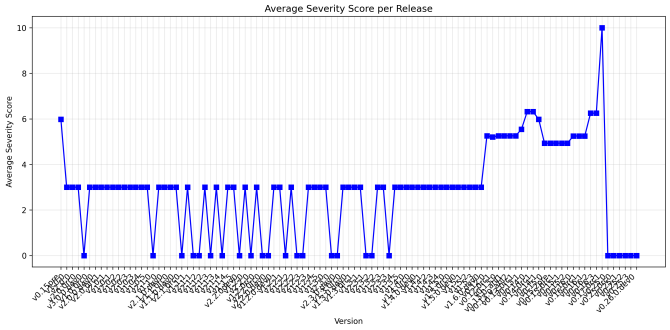


Fig. 16. Vulnerability Timeline

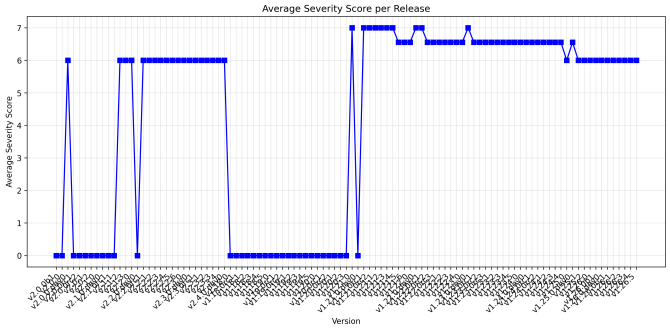


Fig. 17. Vulnerability Timeline

libraries ultimately adds to the reliability and applicability of the model to large-scale, open-source ecosystems.

The figure 16 illustrates the Average Severity Score per Release across multiple versions of a software project. The x-axis indicates software versions (v0.1, v1.0, etc.) and the y-axis captures the corresponding average CVSS (Common Vulnerability Scoring System) severity score that was detected for each release. At the early onset, the average severity scores show slight fluctuations ranging from 0 to 3, representing minor vulnerabilities. Over time the project progresses, the average severity scores ramped dramatically from earlier in the project, specifically toward the later versions (as shown at ”v0.18”) suggesting new and/or increased severity vulnerabilities were either added or rescinding dependencies added to unresolved issues. The upward trend toward the end of the project indicates an increase in severity of vulnerabilities that have occurred in the later releases, which may be due to the dependencies that were up to date and/or more functionalities that had been added over time.

The figure 17 illustrates Average Severity Score per Release, but in a different project or SBOM dataset. Compared to the first graphic, this shows more stable and higher average severity scores (mainly ranging between 6 and 7) in more recent versions, after some fluctuation. Earlier versions of the dataset show low or zero vulnerability scores, followed by a gradual increase in average scores to much higher levels of vulnerability, suggesting that density of vulnerabilities increased as a function of the software being maintained and matured over time. The plateau pattern indicates that likely after some dependencies that contained a higher risk rating were located and adopted into the project, subsequent releases stayed stable with that level of severity. Maintenance at this higher average level of severity, suggests that there are still either the same risks, or no or limited functions were implemented to patch vulnerabilities, in each subsequent release.

B. Changes to make model scalable

We designed our implementation to enhance the model’s scalability and efficiency, by using **parallelism** and **caching mechanisms**. Parallelism allows the simultaneous execution of multiple SBOM generations and vulnerability analysis across different versions of the project, thereby dramatically reducing the overall time to process the projects. It allows for efficient analyses of larger repositories that have a bigger history of commits. Caching acts as a mechanism to store intermediate results (e.g., SBOMs from prior generations or previously fetched vulnerability data) to avoid costly calculations of the data or retrieval from external APIs. These strategies work together to improve performance, better utilize resources, and scalability, so that the model can process larger numbers of projects and versions smoothly.

C. Results

```
Execution Time Summary:
  Sbom Generation          : 317.27 seconds
  Sbom Parsing             : 0.35 seconds
  Vuln Enrichment          : 1332.60 seconds
  Analysis                 : 0.43 seconds
  Visualization            : 12.80 seconds

Total Time: 1663.46 seconds
```

Fig. 18. Full pipeline time

The figure 18 provides an overview of the total run time for 2 projects, and all the versions in each project to be processed. The total run time for SBOM generation was approximately 317 seconds, and vulnerability enrichment again was the most costly phase at 1332 seconds. The remaining phases of parsing, analysis, and visualizations were comparably fast. Overall, the total run time of the pipeline was 1663.46 seconds, indicating that enrichment is the most expensive phase of processing time throughout the pipeline.

Following the scaling of the model using 10 large projects and not just 2 medium-sized projects in the previous experimentation, the benchmarking pipeline as shown in the figure 19 clearly shows a large increase in overall execution time as a result of having more data and complexity. The impact of the model on SBOM generation is considerable; generating the SBOM took the bulk of the runtime at (727.84 seconds), indicating that there was significant dependency extraction occurring across multiple repositories. Positive outcomes affecting performance during SBOM parsing and vulnerability

```
Full Pipeline Completed Successfully!

Execution Time Summary:
 SBOM Generation (all): 727.84 seconds
 SBOM Parsing (flask): 0.04 seconds
 SBOM Parsing (django-rest-framework): 0.06 seconds
 SBOM Parsing (pip): 0.07 seconds
 SBOM Parsing (psycopg2): 0.00 seconds
 SBOM Parsing (pandas): 0.12 seconds
 SBOM Parsing (aiohttp): 0.18 seconds
 SBOM Parsing (requests): 0.08 seconds
 SBOM Parsing (numpy): 0.25 seconds
 SBOM Parsing (pytest): 0.16 seconds
 SBOM Parsing (urllib3): 0.12 seconds
 Vulnerability Enrichment (django-rest-framework): 29.66 seconds
 Vulnerability Enrichment (flask): 30.24 seconds
 Vulnerability Enrichment (numpy): 38.16 seconds
 Vulnerability Enrichment (psycopg2): 0.00 seconds
 Vulnerability Enrichment (pip): 8.44 seconds
 Vulnerability Enrichment (pandas): 16.30 seconds
 Vulnerability Enrichment (aiohttp): 47.10 seconds
 Vulnerability Enrichment (requests): 11.17 seconds
 Vulnerability Enrichment (pytest): 32.40 seconds
 Vulnerability Enrichment (urllib3): 30.03 seconds
 Analysis (flask): 0.03 seconds
 Analysis (django-rest-framework): 0.04 seconds
 Analysis (pip): 0.05 seconds
 Analysis (psycopg2): 0.00 seconds
 Analysis (aiohttp): 0.09 seconds
 Analysis (pandas): 0.09 seconds
 Analysis (numpy): 0.13 seconds
 Analysis (requests): 0.04 seconds
 Analysis (pytest): 0.08 seconds
 Analysis (urllib3): 0.08 seconds
 Visualization (flask): 3.48 seconds
 Visualization (django-rest-framework): 5.25 seconds
 Visualization (numpy): 5.77 seconds
 Visualization (psycopg2): 0.00 seconds
 Visualization (aiohttp): 9.07 seconds
 Visualization (pandas): 5.75 seconds
 Visualization (pip): 6.08 seconds
 Visualization (pytest): 6.40 seconds
 Visualization (requests): 5.01 seconds
 Visualization (urllib3): 5.22 seconds

Total Time: 1025.06 seconds
```

Fig. 19. After scalable

enrichment were still observed, as they both performed relatively well per project, demonstrating aspects of parallelization and caching. Only slight increases in time to complete were observed in the analysis and visualization stages, indicating that the system is running well and demonstrates good scalability per the downstream processing. Overall, the results gave total pipeline time of **1025.06 seconds** which gave confidence that the system performs well with large data sets while replicating reproducibility across each stage.

In summary, comparing the early tests versus the scaled-up tests clearly shows the model's scalability. Even though processing only 2 medium-size projects took 1663.46 seconds previously, the larger-scale processing of 10 large-size projects took less total time (1025.06 sec), despite a more complex dataset and additional size and complexity. The efficiency of the parallelization and caching systems engaged ensures relatively rapid SBOM parsing and enrichment with larger datasets. Overall, the system can be said to be scalable and efficient with larger datasets.

VI. Adaptability Analysis

A. Adaptability Criteria

To increase the generalizability of the project, the scope was widened to not only analyze Python-based projects, but also Java projects. Accommodating this adaptation meant that differences in build systems, such as Maven or Gradle, dependency formats, and SBOM generation tools that work with Java ecosytems needed to be addressed. Ultimately, by allowing for projects that are written in multiple programming languages, the model was able to showcase the ability to support processing and analysis of software over diverse platforms and ecosystems and independently of the programming language for vulnerability detection, thus broadening the flexibility and usability of the framework overall.
Performance metrics are:

- **SBOM Generation Success**: Whether an SBOM was successfully generated for the project.
- **Dependency Extraction Success**: Whether all dependencies were correctly identified and extracted.
- **Vulnerability Mapping Success**: Whether vulnerabilities were successfully mapped to the extracted dependencies.

B. Changes to make model adaptable

Notably, there were no structural or code-level changes made to the original model when testing it on Java projects. The framework was able to generate SBOMs and surface vulnerabilities in the Java-based repositories despite the fact that it was originally developed to analyze Python repositories. This suggests that the model's core pipeline is language agnostic, meaning that it can work across diverse ecosystems with no additional tuning or modification.

C. Results

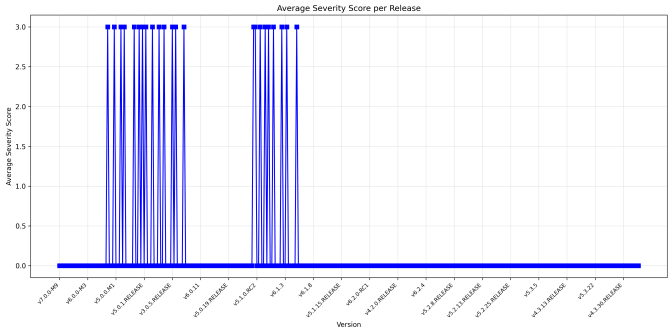


Fig. 20. Average Severity Score

The assessment of the Java project as shown in figure 20 indicates again, our model provided results without code alteration, confirming its language adaptability. The above plot depicts the average severity score per release version for the Java project. The mean vulnerability severity (based on CVSS scores) has been computed for each version. As shown, many of the versions show an average score near 3.0, meaning there is a moderate to high amount of vulnerabilities present. Other

release versions show 0.0 which means that there were no vulnerabilities detected. This shows that our model is also able to identify and quantify vulnerabilities in different versions of Java projects similar to what was done for Python projects. The performance metrics that we used showed 50% correct results for two java projects as well because when we ran the model for 2 java projects which were retro-fit & spring-framework libraries. In summary, comparing the early tests versus the scaled-up tests clearly shows the model's scalability. Even though processing only 2 medium-size projects took 1663.46 seconds previously, the larger-scale processing of 10 large-size projects took less total time (1025.06 sec), despite a more complex dataset and additional size and complexity. The efficiency of the parallelization and caching systems engaged ensures relatively rapid SBOM parsing and enrichment with larger datasets. Overall, the system can be said to be scalable and efficient with larger datasets.

VII. Comparative Study

In this section, we analyze our proposed model against six other research contributions that have comparable concepts and/or methodologies. We have organized our comparison into four dimensions: methodology, evaluation measures, results, and innovation. In analyzing each previous study and their specific approach to SBOM generation, vulnerability analysis, or software security problems, we highlight improvements offered by our model. Lastly, we show our model is innovative because it delivers longitudinal, commit-level security information, as well as multi-format SBOM analysis established through none of the previous models.

A. Comparison with existing works

[8]) Understanding the Effectiveness of SBOM Generation Tools for Manually Installed Packages in Docker Containers (August 2024)

Table 3: Vulnerable Packages that are Overlooked by SBOM Generators and Become Active While Containers were Run with Default or Dummy Parameters

#	Package name	version	# of installations in the dataset	Exemplar vulnerabilities.
1	cachet	2.3.13	3	CVE-2023-43661
2	consul-template	0.12.2	6	CVE-2022-38149
3	fluent-bit	0.12.1	1	CVE-2020-35963
4	grafana	7.2.1	1	CVE-2023-3128
5	logstash-oss	7.10.2	1	CVE-2021-22138
6	nats-server	2.2.6	2	CVE-2022-24450
7	node.js	4.9.1	1	CVE-2018-7158
8	openjdk	8u312	17	CVE-2023-21967
9	owncloud	10.0.10	1	CVE-2021-35846
10	prometheus	1.8.2	2	CVE-2019-3826
11	rabbitmq	3.10.6	1	CVE-2023-46118
12	wordpress	4.9.1	1	CVE-2021-44223

Fig. 21. Existing Paper's Results

Compared to the existing paper as shown in the Figure 21, which typically identifies one vulnerable package with a single version and a single CVE, our model provides much deeper visibility. We detect multiple versions of the same package as shown in the Figure 22 and 23, each potentially carrying multiple vulnerabilities for 2 libraries specifically i.e fluent-bit and logstash, giving a more realistic and comprehensive

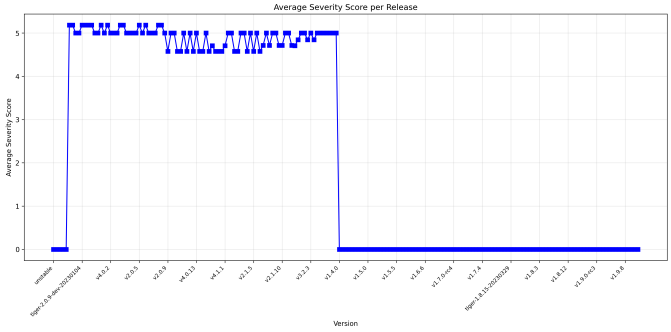


Fig. 22. Fluent-bit library historical analysis

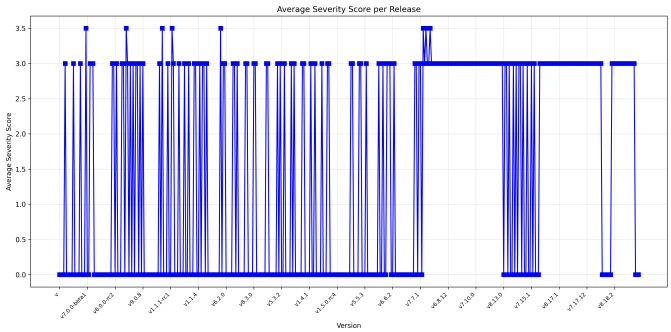


Fig. 23. Logstash library historical analysis

security assessment. This multi-version, multi-CVE analysis uncovers risks that earlier work completely misses, making our approach significantly stronger and more accurate for real-world SBOM evaluation.

Most previous work mostly concentrates on SBOM tools related to containers and measures scarcely utilized packages or scan outcomes, based on primarily a single format. On the other hand, we not only create multi-format (SPDX + CycloneDX) SBOMs, but also provide historical version analysis and track vulnerabilities over time. Sharing knowledge in this way is deeper, unbiased, and security-centric—disclosing real risk trend and not just tool behavior.

[6]) Vulnerability Impact Analysis in Software Project Dependencies Based on Satisfiability Modulo Theories (SMT) (December 2023)

The figure 24 visualizes how the average CVSS severity score changes across software versions. It highlights spikes, drops, and stable periods, helping identify risky and safer releases over time. The previous study is founded on a SMT-based theoretical model that takes a mathematical approach to model dependency graphs and then reason about propagation of vulnerabilities using constraints. In contrast, our model is empirical; instead of performing abstract reasoning about propagation, we make use of real SBOM data, actual changes over time in versions, and the observable evolution of vulnerabilities as shown in the Figure 24. The study does provide metrics for measuring solver *accuracy,*

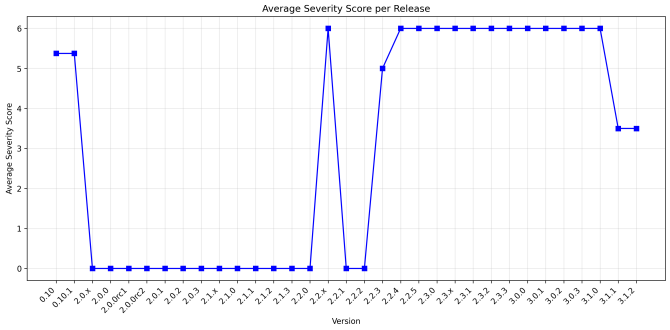


Fig. 24. Historical Commit Analysis

correctness in propagation, and computational performance; whereas our metrics were practical security signals, such as *CVSS trends, vulnerability removal over releases, and high-churn dependencies*. While SMT results demonstrate correctness for static dependency graphs, our framework showcased longitudinal security signals, trajectories for safer versions, and rates of how dependencies evolved (added, removed, or evolved) across commits. Finally, the previous research provided only snapshot based analysis, whereas our model performed historic commit tracing across many versions for a temporal signal of vulnerability evolution and it was a crucial limitation of the SMT-based study.

[1]) Comprehensive Analysis and Recommendation of Supply Chain Risk Management Framework for the Military Domain (May 2025)

```
Execution Time Summary:
 SBOM Generation (all): 727.84 seconds
 SBOM Parsing (flask): 0.04 seconds
 SBOM Parsing (django-rest-framework): 0.06 seconds
 SBOM Parsing (pip): 0.07 seconds
 SBOM Parsing (psycopg2): 0.00 seconds
 SBOM Parsing (pandas): 0.12 seconds
 SBOM Parsing (aiohttp): 0.18 seconds
 SBOM Parsing (requests): 0.08 seconds
 SBOM Parsing (numpy): 0.25 seconds
 SBOM Parsing (pytest): 0.16 seconds
 SBOM Parsing (urllib3): 0.12 seconds
```

Fig. 25. Real World Implementation

The Figure 25 reports the total SBOM generation time and per-project SBOM parsing times. It demonstrates the performance efficiency of the pipeline and the relative cost of analyzing different repositories. The paper cited presents a framework, identifies types of information, suggests a 6-phase C-SCRM model for evaluation, and has practitioners give input on it, thus it is conceptual, not empirical. Our model on the other hand, is a data driven model using SBOM generation on a commit level to track vulnerabilities over time and dependencies over version updates in real software as shown in the Figure 25. Their measurements *assess the components for completeness or conforming to a standard*, while we measure the actual security behaviour by quantifying *vulnerability additions/removals, CVSS trends, and changes in severity*. Their findings emphasize information missing from U.S. frameworks, while our findings offer recommendations from security intelligence based on SBOM history, vulnerability timelines, and recommending a safer version. Finally, their paper is snapshot based and has limited generalizability in software supply chain context; our model gets around that by analyzing historical commits and empirical evaluations of actual software projects (e.g., pandas, numpy), resulting in a time-series and system validated, empirical realization of risk from the software supply chain.

[2]) On the Adoption of Software Bill of Materials in Open-Source Software Projects (June 2025)

	A	B
	CycloneDX Diff	SPDX Diff
	Added: ['actions/setup-python', 'ac	Added: ['actions/setup-python', '
	Added: ['pytest-mock', 'httpbin'], R	Added: ['pytest-mock', 'httpbin'],
	Added: [], Removed: ['actions/setu	Added: [], Removed: ['actions/se
	Added: [], Removed: []	Added: [], Removed: []
	Added: ['actions/setup-python', 'ac	Added: ['actions/setup-python', '
	Added: ['pytest-mock', 'httpbin'], R	Added: ['pytest-mock', 'httpbin'],
	Added: ['slsa-framework/slsa-githu	Added: ['slsa-framework/slsa-git
	Added: [], Removed: ['slsa-framew	Added: [], Removed: ['slsa-frame
)	Added: [], Removed: ['actions/setu	Added: [], Removed: ['actions/se

Fig. 26. CycloneDx and Spdx

The image 26 shows side-by-side component-level diffs generated using CycloneDX and SPDX formats. Both formats capture added and removed dependencies, demonstrating consistency in multi-format SBOM comparison. The paper you mentioned investigates GitHub repositories to determine the existence of SPDX or CycloneDX SBOMs. It looks at project repositories and reviews the completeness of samples, but ultimately it remains a static metadata audit without studying changes in dependencies or evolution of dependencies. In contrast, our model produces SBOMs across multiple commits, compares SBOM versions as shown in the Figure 26, and builds histories of vulnerabilities over time as shown in the Figure 24, providing genuinely actionable security data. The paper does use *metrics in relation to SBOM use, schema, and completeness*, while our metrics are descriptive of real security outcomes such as *additions/removals of vulnerabilities, trends in severity, and progress of CVSS*. The paper presents relatively low and inconsistent SBOM usage, but has no insight into the behaviour of security at the project level. Our results, on the other hand provide the opportunity for commit based patterns of vulnerabilities, dependency churn, CVSS trajectories and the ability to identify safer versions. Finally, the paper has a limitation of bias due to format, and the static snapshot analysis of data, while our model improves on this by examining historical commits to produce several SBOMs and comparing across formats to improve applicability.

TABLE III
COMPARISON OF EXISTING WORK WITH OUR MODEL

Sr. No	Existing Methodology	Comparison	Existing Metrics	Metrics Comparison	Existing Results	Results Comparison	Limitations
[8]	Empirical analysis of SBOM tools on Docker containers	Our model provides historical SBOM + vulnerability evolution	Overlooked packages, vulnerability presence	Our metrics show real security outcomes	Tools vary in detection coverage	Our model gives actionable version-wise security	Tool bias + snapshot analysis
[6]	SMT modeling and vulnerability propagation across dependency graphs	Our model is empirical, commit-wise tracking	Solver accuracy, propagation correctness	Our metrics reflect real vulnerability change	Precise propagation modelling	Our model shows real risk shifts over versions	Snapshot + scalability issues
[1]	Derived 32 info types → proposed 6-phase C-SCRM framework	Our model provides empirical SBOM timelines	Coverage, conformance, practitioner feedback	Our metrics quantify actual security change	Framework completeness improved	Our results give measurable security evolution	Snapshot + limited generalizability
[2]	Mined GitHub for SBOM files (186 projects)	Our model generates SBOMs across commits	Adoption rate, storage, completeness	Our metrics measure vulnerabilities, CVSS trends	SBOM adoption low, inconsistent quality	Our model outputs project-specific security trends	Tool dependency + snapshot
[3]	Blockchain-based SBOM system (Hyperledger Fabric)	Our model focuses on real software security	Latency, accuracy, throughput, cost	Our metrics show version-wise vulnerability changes	Real-time detection in blockchain system	Our results show long-term risk movement	Limited real-world evaluation
[4]	AI-driven risk + conformity model (LSTM, CNN)	Our model studies vulnerability evolution, not ML accuracy	Accuracy, F1, TPR/FPR, exposure reduction	Our metrics capture real software risk	High anomaly detection accuracy	Our results show safer-version and CVSS trends	Domain-limited + tool dependency

[3]) Blockchain-Integrated Software Bill of Materials (SBOM) for Real-Time Vulnerability Detection in Decentralized Package Repositories (May 2023)

The study mentioned creates a blockchain-supported SBOM framework by using Hyperledger Fabric, smart contracts, and automated CVE feeds, with an emphasis on decentralized storage, trust authentication, and system performance. Our approach creates an SBOM at a commit-level and tracks actual vulnerability behavior over software releases with an emphasis on the evolving security rather than mechanics as shown in the Figure 24. The study provides insight into detection *latency, accuracy, throughput and overhead* from using a blockchain transaction, while my data provides insight into *vulnerabilities added or removed per version, CVSS trends, severity evolution, and dependency turnover*. The paper validates the evaluation of a distributed SBOM system's scalability and correctness, but does not provide insight into how software security evolves. My study uses real-world vulnerability timelines, identifying safer versions of software and tracking the evolution of dependencies from version to version. Finally, while the blockchain research relies on controlled/simulated environments to gather results, my study analyzes real-world projects, addressing the generalizability issues and providing relevant security insights based on real-world software ecosystems.

[4]) Analyzing FOSS License Usage in Publicly Available Software at Scale via the SWH-Analytics Framework (April 2024)

The paper in question offers an analysis of patterns in FOSS licensing across 835 GitHub user repositories leveraging Software Heritage Analytics and ScanCode. It examined declared versus in-code licenses, various structures for multiple licenses, and correlations that achieved statistical significance with respect to compliance. Our work emphasizes understanding real security behavior by producing software Bill of Materials (SBOMs) from multiple commits as shown in the Figure 24, observing the growth of dependencies, and measuring changes in vulnerabilities across versions of software. The metrics from this paper emphasized *license counts, license conflicts, correlations, and statistical distributions*. In comparison, our metrics captures the differences in *vulnerabilities added or removed version-wise, CVSS trends, movements in severity level, and dependency churn offering* security-oriented findings. By presenting findings as license diversity and compliance risk, the paper does not provide visibility into how a project's security changes over time. Our findings instead present actionable findings based on changes in vulnerabilities timelines, safer-versions of dependencies, and evolution of dependencies. Finally, while this paper synthesizes pull-requests' functionality from snapshots and one

tool to present findings, we analyze historical commits through SBOMs to develop a dynamic longitudinal perspective of actual software security.

Table III provides a summary of six notable research to study and compare the research on two levels–the methodology used, metrics employed, corresponding results, and limitations, in comparison to our proposed model. The existing studies are related to SBOM tools, SMT-based dependency reasoning, a supply-chain risk management framework, the SBOM adoption study, a model for blockchain-based SBOM verification, and an artificial intelligence-based risk assessment tool. This comparison emphasizes the fact that the methodologies of the studies are all either static, dependent on a tool, or rely on an infrastructure; our proposed model is empirical because it creates SBOMs at the commit level, while vulnerability data reflects the real evolution of vulnerability over time. The metrics column likewise contrasts framework-based metrics and performance-level metrics to the metrics of an outcome-driven focused on security in the model we propose and the other studies. The results comparison underlines that the other studies focus either on tool behaviour, or metrics focused on evaluating how efficient the systems were, not on how they improved security. Lastly, the limitations section highlights shortcomings common to all, such as snapshot based analysis, bias, limited generalizability; all issues our proposed model solves through SBOM longitudinal analysis and real-world project evaluation.

VIII. Conclusion & Future Scope

A. Conclusion

This paper proposed an overarching SBOM-centric framework for assessing software supply chain security through commit-based vulnerability dynamics. By constructing SBOMs across each version and enriching them with feeds from OSV and the NVD, the model offered insight into dependency changes, severity trends, and longitudinal security behavior. The analysis of historical data demonstrated that historical assessments captured longitudinal trends that were missed in studies that take snapshots such as repeated high-severity vulnerabilities in early versions of Flask and spikes in dependency churn rates in versions 2.2.x and 3.x. Experiments conducted on Python and Java repositories confirmed that the pipeline is agnostic to programming languages and adaptable without changes to structure. Scalability tests resulted in a significant decrease in total execution time while assessing 10 large Java and Python projects, further validating the efficiencies of parallelization and caching strategies. Altogether, the model achieved results that signify actionable security insights including safer versions of dependencies, high-risk dependency patterns, and SBOM-based security intelligence across the width of the software and supply chain lifecycle.

B. Future Scope

There are many ways to grow this framework. Firstly, the addition of static and dynamic analysis with SBOM-based vulnerability data may offer a more holistic risk score. Secondly, the addition of real-time CI/CD hooks to automatically update SBOMs and track vulnerabilities in enterprise pipelines could be achieved. Thirdly, advanced ML models to assess the likelihood of vulnerability introduction based on commit metadata, dependency age, or ecosystem trends could be included. Fourth, better support for other ecosystems, including Rust, Go, and container-native builds, would improve application. Finally, a dashboard to visualize real-time supply chain risks would facilitate organizations' ability to operationalize findings in enterprise DevSecOps scales.

References

[1] Ahn, Jung, Cho, Kwangsoo, Seo, Kyungdeok, Kim, Hyun-ji, Kim, Seungjoo, Comprehensive Analysis and Recommendation of Supply Chain Risk Management Framework for the Military Domain, *IEEE Access*,2025.

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

[3] B. S. Adelusi, et al., "Blockchain-Integrated Software Bill of Materials (SBOM) for Real-Time Vulnerability Detection in Decentralized Package Repositories," *International Journal of Scientific Research in Computer Engineering*, vol. 7, no. 3, pp. 45–52, 2023.

[4] K. Kioskli, et al., "A Risk and Conformity Assessment Framework to Ensure Security and Resilience of Healthcare Systems and Medical Supply Chain," *Journal of Healthcare Informatics*, vol. 10, no. 2, pp. 101–120, 2025.

[5] Antelmi, A., Torquati, M., Corridori, G. et al. Analyzing FOSS license usage in publicly available software at scale via the SWH-analytics framework. J Supercomput 80, 15799–15833 (2024). https://doi.org/10.1007/s11227-024-06069-x

[6] A. Germán Márquez, Ángel Jesús Varela-Vaca, María Teresa Gómez López A dataset on vulnerabilities affecting dependencies in software package managers *Elsevier*, 2025.

[7] Eman Abu Ishgair and Marcela S. Melara and Santiago Torres-Arias SoK: A Defense-Oriented Evaluation of Software Supply Chain Security *arXiv*, 2024, 2405.14993

[8] Nobutaka Kawaguchi and Charlie Hart and Hiroki Uchiyama, Understanding the Effectiveness of SBOM Generation Tools for Manually Installed Packages in Docker Containers *Journal of Internet Services and Information Security*, vol. 14, no. 3, pp. 191–212, 2024.

[9] Bi, Tingting and Xia, Boming and Xing, Zhenchang and Lu, Qinghua and Zhu, Liming On the Way to SBOMs: Investigating Design Issues and Solutions in Practice *Association for Computing Machinery*, vol 33 number 6, July 2024.

[10] Alberto Tacchella, Emanuele Beozzo, Bruno Crispo, and Marco Roveri Firmware Secure Updates meet Formal Verification *ACM Trans. Cyber-Phys. Syst*, 2025.

[11] Menghan Wu, Yukai Zhao, Xing Hu, Xian Zhan, Shanping Li, and Xin Xia More Than Meets the Eye: On Evaluating SBOM Tools In Java *ACM Trans. Softw. Eng. Methodol.*, 2025.

[12] Benedetti, G., Cofano, S., Brighente, A., Conti, M. (2025). The Impact of SBOM Generators on Vulnerability Assessment in Python: A Comparison and a Novel Approach. In: Fischlin, M., Moonsamy, V. (eds) Applied Cryptography and Network Security. *ACNS 2025. vol 15826. Springer, Cham*.

[13] A landscape study of open source and proprietary tools for software bill of materials (sbom), Mirakhorli, Mehdi and Garcia, Derek and Dillon, Schuyler and Laporte, Kevin and Morrison, Matthew and Lu, Henry and Koscinski, Viktoria and Enoch, Christopher, *arXiv*, 2024.

[14] Yu, Sheng and Song, Wei and Hu, Xunchao and Yin, Heng On the correctness of metadata-based SBOM generation: A differential analysis approach *IEEE* 2024.

[15] Fucci, Davide and Di Penta, Massimiliano and Romano, Simone and Scanniello, Giuseppe, Augmenting Software Bills of Materials with Software Vulnerability Description: A Preliminary Study on GitHub, *Association for Computing Machinery*, 2025.

[16] L. Soeiro, T. Robert and S. Zacchiroli, "Wild SBOMs: a Large-scale Dataset of Software Bills of Materials from Public Code,", *IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR), Ottawa, ON, Canada*, 2025.

[17] S. Tanzarella and M. Repetto, "Context Discovery for Digital Service Chain with OpenC2," *IEEE 11th International Conference on Network Softwarization (NetSoft), Budapest, Hungary*, 2025.

[18] W. Otoda, T. Kanda, Y. Manabe, K. Inoue and Y. Higo, "SBOM Challenges for Developers: From Analysis of Stack Overflow Questions," *IEEE/ACIS 22nd International Conference on Software Engineering Research, Management and Applications (SERA), Honolulu, HI, USA*, 2024.

[19] D. Garcia et al., "A Landscape Study of Open-Source Tools for Software Bill of Materials (SBOM) and Supply Chain Security," *IEEE/ACM 3rd International Workshop on Software Vulnerability Management (SVM), Ottawa, ON, Canada*, 2025.

[20] C. Yang-Smith and A. Abdellatif, "Tracing Vulnerabilities in Maven: A Study of CVE lifecycles and Dependency Networks," 2025 *IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR), Ottawa, ON, Canada*, 2025.

[21] S. H. B. I. Kumar, L. R. Sampaio, A. Martin, A. Brito and C. Fetzer, "A Comprehensive Study on the Impact of Vulnerable Dependencies on Open-Source Software," *IEEE 35th International Symposium on Software Reliability Engineering (ISSRE), Tsukuba, Japan*, 2024.

[22] E. O'Donoghue, A. M. Reinhold and C. Izurieta, "Assessing Security Risks of Software Supply Chains Using Software Bill of Materials," *IEEE International Conference on Software Analysis, Evolution and Reengineering - Companion (SANER-C), Rovaniemi, Finland*, 2024.

[23] Y. Zhao, Y. Zhang, D. Chacko and J. Cappos, "CovSBOM: Enhancing Software Bill of Materials with Integrated Code Coverage Analysis," *IEEE 35th International Symposium on Software Reliability Engineering (ISSRE), Tsukuba, Japan*, 2024.

[24] Shen, Y., Gao, X., Sun, H. et al. Understanding vulnerabilities in software supply chains. *Empirical Software Engineering*, 2025.

[25] Wu, Menghan and Zhao, Yukai and Hu, Xing and Zhan, Xian and Li, Shanping and Xia, Xin, More Than Meets the Eye: On Evaluating SBOM Tools In Java, *Association for Computing Machinery*, 2025.

[26] Yousefnezhad, N., Costin, A. (2024). Understanding SBOMs in Real-World Systems – A Practical DevOps/SecOps Perspective. In: Shishkov, *B. (eds) Business Modeling and Software Design. BMSD*, 2024.

[27] Can Ozkan and Xinhai Zou and Dave Singelee, Supply Chain Insecurity: The Lack of Integrity Protection in SBOM Solutions, *arxiv*, 2025.

[28] S. Nocera, M. Di Penta, R. Francese, S. Romano and G. Scanniello, "If it's not SBOM, then what? How Italian Practitioners Manage the Software Supply Chain," *IEEE International Conference on Software Maintenance and Evolution (ICSME), Flagstaff, AZ, USA*, 2024.

[29] A. Lemay and N. Katiyar, "Supply Chain Risk Analysis Via SBOM Data Enrichment," *IEEE International systems Conference (SysCon), Montreal, QC, Canada*, 2025

[30] L. S. Beevi, P. R. K. S. Bhama, J. A. Vijayan, J. P. P. M., W. V. Dani and J. Premalatha, "Optimizing Supply Chain Security Using ACO and Blockchain: Integrating Root of Trust, Unclonable Functions, and SBOM for Cybersecurity," *International Conference on Visual Analytics and Data Visualization (ICVADV), Tirunelveli, India*, 2025.

[31] J. Kim, Y. Choi and S. Kim, "Attestation-Based SBOM Integrity Verification for Secure and Transparent Software Supply Chains," *25th Asia-Pacific Network Operations and Management Symposium (APNOMS), Kaohsiung, Taiwan*, 2025.

[32] Kloeg, Berend and Ding, Aaron Yi and Pellegrom, Sjoerd and Zhauniarovich, Yury, Charting the Path to SBOM Adoption: A Business Stakeholder-Centric Approach, *Association for Computing Machinery*, 2024.

[33] Bi, Tingting and Xia, Boming and Xing, Zhenchang and Lu, Qinghua and Zhu, Liming, On the Way to SBOMs: Investigating Design Issues and Solutions in Practice, *Association for Computing Machinery*, 2024.

[34] R. Kishimoto, T. Kanda, Y. Manabe, K. Inoue and Y. Higo, "Osmy: A Tool for Periodic Software Vulnerability Assessment and File Integrity Verification using SPDX Documents," *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), Rovaniemi, Finland*, 2024.

[35] Eman Abu Ishgair and Chinenye Okafor and Marcela S. Melara and Santiago Torres-Arias, Trustworthy and Confidential SBOM Exchange Eman Abu Ishgair and Chinenye Okafor and Marcela S. Melara and Santiago Torres-Arias, *arxiv*, 2025.

[36] S. Cofano, G. Benedetti and M. Dell'Amico, "SBOM Generation Tools in the Python Ecosystem: an In-Detail Analysis," *IEEE 23rd International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), Sanya, China*, 2024

[37] J. ". Rhee, F. Zuo, C. Tompkins, J. Oh and Y. R. Choe, "GrizzlyBay: Retroactive SBOM with Automated Identification for Legacy Binaries," *IEEE Military Communications Conference (MILCOM), Washington, DC, USA*, 2024

[38] N. Kawaguchi and C. Hart, "On the Deployment Control and Runtime Monitoring of Containers Based on Consumer Side SBOMs," *IEEE 21st Consumer Communications & Networking Conference (CCNC), Las Vegas, NV, USA*, 2024

[39] Fernando Vera and Palina Pauliuchenka and Ethan Oh and Bai Chien Kao and Louis DiValentin and David A. Bader, Profile of Vulnerability Remediations in Dependencies Using Graph Analysis, *arxiv*, 2024.