CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Framework Insights and Automated Attestation for Software Supply Chain Security

Master's thesis in Computer science and engineering

Elias Falk, Emil Svensson

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2025

Master's thesis 2025

Framework Insights and Automated Attestation for Software Supply Chain Security

Elias Falk, Emil Svensson

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2025

Framework Insights and Automated Attestation for Software Supply Chain Security

Elias Falk, Emil Svensson

© Elias Falk, Emil Svensson 2025.

Supervisor: Md Masoom Rabbani, Department of Computer Science and Engineering
Advisor: Francisco Blas Izquierdo Riera, KITS AB
Examiner: Andrei Sabelfeld, Department of Computer Science and Engineering

Master's Thesis 2025
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Gothenburg, Sweden 2025

Framework Insights and Automated Attestation for Software Supply Chain Security

Elias Falk, Emil Svensson
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

The escalating threat of software supply chain attacks necessitates robust security measures; however, current guidance is fragmented across numerous, often overlapping, frameworks. This thesis addresses this challenge through a dual approach. First, it conducts a systematic comparative analysis of five prominent software supply chain security frameworks - ESF, S2C2F, SCVS, SLSA, and an academic SOK taxonomy - by decomposing their 284 guidelines into 1,321 atomic, actionable statements. These statements were then thematically labeled and semantically compared to define framework scope, identify consensus areas, reveal gaps, and highlight specialized strengths. The analysis found ESF to be the most comprehensive, while S2C2F, SCVS, and SLSA offer significant depth in specific niches, such as consumption, component verification, and build integrity, respectively. This underscores that no single framework is universally optimal.

Second, this research develops and evaluates a Proof-of-Concept (PoC) system to demonstrate the feasibility of automating compliance attestation. The PoC automatically verifies a targeted subset of decomposed guidelines for selected open-source projects, embedding cryptographically signed conformance attestations - including claims, evidence, and targets - directly within a Software Bills of Materials (SBOM). A companion visualization tool enables human inspection and signature verification of these enriched SBOMs. A feasibility study confirmed the viability of this end-to-end process, showcasing a practical pathway for integrating verifiable compliance into the software development lifecycle. Ultimately, this work provides a clearer map of the current guidance landscape and demonstrates a practical path to embedding verifiable compliance, advancing the automation and trustworthiness of software supply chain security.

Keywords: Cybersecurity, Software Supply Chain, Software Supply Chain Security, SBOM, Security Frameworks, SSCS Guidelines, CycloneDX, S2C2F, SLSA, ESF, SCVS

Acknowledgements

We would like to thank our supervisors, Md Masoom Rabbani and Francisco Blas Izquierdo Riera, for their invaluable guidance and support throughout this project. We also extend our thanks to the entire KITS team for their collaboration and to our examiner, Andrei Sabelfeld, for his time and constructive feedback. Finally, our appreciation goes to everyone else who helped and supported us.

Elias Falk, Emil Svensson
Gothenburg, June 2025

Contents

List of Figures xiii

List of Tables xv

1 Introduction 1
1.1 Related Work . . . 2
1.1.1 Semantic Analysis in Security and Policy Contexts . . . 3
1.1.2 Policy Extraction and Decomposition . . . 3
1.1.3 Frameworks for Build Process Integrity and Attestation . . . 3
1.1.4 Automated Compliance and Security Assessment Tools . . . 4
1.1.5 Integrated and Standardized Compliance Approaches . . . 4
1.2 Aim . . . 4

2 Background 5
2.1 The Software Supply Chain . . . 6
2.2 Software Bill of Materials (SBOM) . . . 6
2.2.1 Definition and Purpose of SBOMs . . . 7
2.2.2 Key SBOM Standards: SPDX and CycloneDX . . . 7
2.2.3 SBOM Generation and Tooling . . . 7
2.2.4 SBOM Signing and Integrity Verification . . . 8
2.2.5 CycloneDX Attestation Model . . . 8
2.3 Software Supply Chain Security Recommendations . . . 9
2.3.1 Enduring Security Framework (ESF): Recommended Practices Guide for Developers . . . 9
2.3.2 The Secure Supply Chain Consumption Framework (S2C2F) . 10
2.3.3 Software Component Verification Standard (SCVS) . . . 11
2.3.4 Supply-chain Levels for Software Artifacts (SLSA) . . . 11
2.3.5 Systematization of Knowledge (SOK): Taxonomy of Attacks on Open-Source Software Supply Chains . . . 12

3 Methods 15
3.1 Comparative Analysis . . . 16
3.1.1 Framework Selection . . . 16
3.1.2 Framework Analysis and Guideline Representation . . . 18
3.1.3 Decomposition of Guidelines . . . 18

3.1.3.1 Extraction of Guidelines 18
3.1.3.2 Decomposition Process 19
3.1.4 Semantic Comparison of Decomposed Guidelines 20
3.1.5 Labeling of Decomposed Guidelines 21
3.1.5.1 Initial Category Formation 22
3.1.5.2 Iterative Sampling and Category Refinement 22
3.1.5.3 Final Application to the Full Corpus 22
3.2 Automated SBOM-Based Compliance Attestation 23
3.2.1 Structuring Conformance Evidence within SBOMs for the PoC System 24
3.2.1.1 Requirement 24
3.2.1.2 Target 25
3.2.1.3 Assessor 26
3.2.1.4 Evidence (E) 27
3.2.1.5 Counter-Evidence (CE) 29
3.2.1.6 Counterclaim 30
3.2.2 Leveraging Decomposed Guidelines for Automated Checks . . 31
3.2.3 Automatic Compliance Checks 32
3.2.3.1 Targeted Guidelines and Checker Mapping for PoC . 33
3.2.3.2 Calculation of Conformance and Confidence Scores . 34
3.2.3.2.1 Conformance Score 35
3.2.3.2.2 Confidence Score 35
3.2.3.3 Check Implementation Details 35
3.2.4 Signing Attestations for Integrity and Authenticity 38
3.2.4.1 Threat Model 38
3.2.4.2 Implementation of Signing using JSON Web Signatures 40
3.2.5 Execution Flow 41
3.2.6 Design of an SBOM Attestation Viewer and Verifier 41
3.2.7 Feasibility Study Experimental Setup 47
3.2.7.1 Target Projects 47
3.2.7.2 Selected Guidelines for Automation 47

4 Results **49**
4.1 Comparative Analysis 50
4.1.1 Decomposition Results 50
4.1.1.1 Decompositions per Framework 50
4.1.1.2 Text Character Counts of Guidelines 51
4.1.2 Semantic Overlap of Decompositions 54
4.1.3 Labeling Decompositions 55
4.1.3.1 Labels 56
4.1.3.2 Label Coverage Across Frameworks 57
4.1.3.3 Distribution of Number of Labels per Decomposition 58
4.1.4 Framework Coverage and Suitability Analysis 59
4.2 Feasibility of the End-to-End Attestation Process 61
4.2.1 Generation and Structuring of Compliance Attestations by the PoC System 61

4.2.1.1 Observations on Selected Guideline Checks 62
4.2.2 Verification and Presentation of Embedded Attestations 63
4.2.3 Assessment of Overall Feasibility 63

5 Discussion 65
5.1 Discussion of Comparative Analysis 66
5.1.1 Insights and Challenges in Guideline Decomposition 66
5.1.1.1 The Variable Nature of Source Guidelines: Length and Granularity 66
5.1.1.2 Challenges in the Decomposition Process 67
5.1.1.2.1 The Human Factor in Manual Decomposition 67
5.1.1.2.2 Handling Logical Structures: The AND/OR Dilemma 67
5.1.2 Evaluating Semantic Relationships Between Frameworks . . . 67
5.1.2.1 Observed Inter-Framework Overlap 68
5.1.2.2 Performance and Limitations of the Semantic Similarity Model 68
5.1.3 Thematic Analysis: Labeling, Distribution, Consensus, and Gaps 69
5.1.3.1 Label Distribution 69
5.1.3.2 Consensus and Gaps of Guidance 70
5.1.3.3 Limitations of the Labeling Approach 71
5.1.3.4 Unlabeled Decompositions 72
5.1.4 Towards Unification: Synthesizing Findings and Proposing Pathways 73
5.1.4.1 Bringing Frameworks To The Same Level of Granularity 73
5.1.4.2 From Static PDFs To An Interactive Portal 73
5.1.4.3 Introducing Maturity Levels and Attack Mappings . 73
5.1.4.4 Tooling Support For Adoption 74
5.2 Discussion of PoC Compliance Tool 75
5.2.1 Feasibility and Limitations of Automating SSCS Guideline Verification 75
5.2.2 SBOM Editing Tool 76
5.2.3 Limitations of the Automated Checks 77

6 Conclusion 79
6.1 Future Work 80

Bibliography 81

List of Figures

2.1 Hierarchical structure of an attestation with its supporting elements and targets. 9

3.1 Overview of the steps taken in the comparative analysis. 16
3.2 High-level diagram of the PoC system. 23
3.3 Example overview display in the visualization tool (addresses FR2). . 43
3.4 Example detailed expanded view for an SCVS requirement (addresses FR3). 44
3.5 Example modal window displaying target component details (addresses FR3). 45
3.6 Example tool dialog confirming successful signature verification (addresses FR4). 45
3.7 Example tool dialog indicating issues with signature verification (addresses FR4). 46
3.8 Example of integrated signature verification status in the requirement view (addresses FR4). 46

4.1 Box plot of the number of decompositions per guideline across frameworks. 51
4.2 Box plot of guideline text character counts across frameworks. 52
4.3 Box plot of decomposed text character counts across frameworks. . . 52
4.4 Coverage of source framework guidelines by target frameworks at different thresholds for cosine similarity. 54
4.5 Heatmap of the label distribution across frameworks. 58

List of Tables

3.1 Guidelines targeted by PoC and their assigned labels 33
3.2 Mapping of guidelines to implemented PoC checkers 34

4.1 Summary of the number of guidelines and their decompositions for each software supply chain framework. 50
4.2 The top 10 most semantically similar statements (cosine similarity). . 55
4.3 The labels and their corresponding description. 56
4.4 The label distribution across frameworks 59
4.5 SSCS framework coverage, focus, and suitability 59
4.5 SSCS framework coverage, focus, and suitability (continued) 60
4.6 Summary of automated compliance check outcomes for the selected projects. (first number = Conformance Score, Cnf. = Confidence Score) 62

1

Introduction

As organizations increasingly depend on third-party components, open-source libraries, and cloud-based build infrastructures, the risk of supply chain attacks has rapidly increased [1], [2]. The consequences of insecure software supply chains extend beyond technical concerns, leading to substantial financial losses, reputational damage, and heightened security risks for organizations [3]. A report from Sonatype highlights this growing threat, revealing that software supply chain attacks experienced an average annual growth rate of 742% between 2020 and 2022 [4]. Malicious actors exploit vulnerabilities in dependencies, compromise build systems, and introduce malicious code into widely used software, leading to widespread security incidents. Notable incidents, such as the SolarWinds attack [5] and the Log4j vulnerability [6], have demonstrated how weaknesses in the supply chain can have cascading effects across industries and organizations worldwide.

In response to this escalating threat, numerous frameworks around best-practice guidelines have been proposed by stakeholders across industry, government, and academia [7]–[11]. Each initiative addresses software supply chain security from a different angle. For example, community and industry efforts have produced standards like OWASPs Software Component Verification Standard (SCVS) and OpenSSF's Secure Supply Chain Consumption Framework (S2C2F), which prescribe a set of controls and practices for managing third-party components and dependencies [7], [9]. Another cross-industry collaboration effort, Supply-chain Levels for Software Artifacts (SLSA), introduces a tiered model of graduated integrity guarantees for build and deployment processes [8]. At the same time, government agencies such as the NSA and CISA have issued comprehensive guidance, most notably, the Enduring Security Framework (ESF) *Securing the Software Supply Chain: Recommended Practices for Developers* to harden the software development lifecycle end-to-end [10]. Academic research has further contributed a systematic perspective, exemplified by a recent Systematization of Knowledge (SOK) that presents a broad taxonomy of over one hundred attack vectors on open-source supply chains and maps them to corresponding mitigations [11].

However, this collection of guidance remains fragmented and inconsistent. The frameworks vary widely in scope and detail. SLSA, for instance, focuses narrowly on build pipeline integrity, whereas SCVS and S2C2F enumerate more extensive control checklists covering everything from component inventory to build environments [7]–[9]. There is overlap in the recommendations they provide, yet each uses its own ter-

minology and priorities, and no single framework offers a holistic view encompassing all known supply chain attack vectors [12]. This fragmentation makes it challenging for developers, organizations, and policy-makers to discern which guidance to follow or how to reconcile multiple standards. Thus, a systematic comparative analysis of these frameworks is needed to illuminate their relative strengths, weaknesses, and gaps. By critically examining the content and level of detail of SCVS, S2C2F, SLSA, the ESF developer guidelines, and the academic taxonomy, this thesis will provide a comprehensive overview of the current supply chain security framework landscape. Such an analysis is both academically valuable and practically important: it can clarify areas of consensus and divergence within existing recommendations, support developers and organizations in navigating the fragmented framework landscape, and contribute to the refinement and alignment of future standards for software supply chain security.

In addition to various existing guidelines, tools such as the OpenSSF Scorecard [13] and Legitify [14] contribute significantly by automatically assessing general software development best practices. However, a notable gap remains: these types of tools typically do not evaluate conformance against the recommendations of established software supply chain security frameworks. However, a notable gap remains: these types of tools typically do not evaluate conformance against the recommendations of established software supply chain security frameworks. This limitation is also apparent in the context of the Software Bill of Materials (SBOM), which serves as an inventory of software components and their dependencies. While SBOMs are becoming increasingly important, a standardized mechanism for automatically embedding conformance attestations within them is largely absent, despite emerging recognition of the potential for formats like CycloneDX to accommodate this [15]. This shortcoming represents an important missed opportunity, especially considering the growing regulatory demands, such as the EUs Cyber Resilience Act, which mandates that organisations maintain an SBOM [16]. Therefore, there is a compelling opportunity to operationalize these existing SBOM functionalities more systematically and leverage all capabilities within an SBOM. We argue that the SBOM, already a regulatory standard, should be systematically employed as a component manifest and a verifiable record of adherence to critical Software Supply Chain Security (SSCS) framework guidelines.

1.1 Related Work

This section reviews key research and development efforts relevant to the aims of this thesis, particularly in the areas of semantic analysis for security, automated policy extraction, frameworks for build process integrity, automated compliance assessment tools, and integrated compliance approaches. These areas inform the methodology and highlight the contributions of this work.

1.1.1 Semantic Analysis in Security and Policy Contexts

Natural language processing (NLP) techniques are increasingly applied to structure and analyze security-related text. For instance, sentence-embedding models like BERT have been employed to map vulnerability descriptions to Common Weakness Enumeration (CWE) categories, thereby improving the triage process by automating the linkage of CVEs to weakness taxonomies [17]. Similar embedding-based methods facilitate the cross-walking of controls across different security standards (e.g., ISO 27001, NIST CSF, MITRE ATT&CK), which can significantly reduce the manual workload involved in policy alignment and comparison [18].

1.1.2 Policy Extraction and Decomposition

Knowledge-extraction systems like Text2Policy demonstrate over 80% accuracy in mining access-control rules from requirement documents, translating natural-language policies into enforceable specifications [19]. Ding *et al.* [20] introduce a two-step prompt learning event extraction framework (TPEE) for Chinese policy texts that achieves F1-scores of 73.92% and 72.81% in argument recognition and classification, demonstrating the power of prompt-based semantic analysis for efficiently extracting structured requirements from complex policy documents. These works highlight the potential of semantic analysis for efficiently extracting structured requirements, a concept related to decomposing and interpreting the guidelines studied in this thesis.

Wanner *et al.* [21] study how to split complex claims into smaller, more precise assertions. They introduce *DecompScore*, a new metric that gauges how well a decomposition captures subclaims actually implied by the original text, and they develop an LLM-driven approach that outperforms previous techniques in both fidelity and granularity. Their method produces the highest count of supported subclaims among those evaluated, though both automated and manual reviews reveal that even the best systems miss some assertions and occasionally combine multiple ideas into one, suggesting further work is needed.

1.1.3 Frameworks for Build Process Integrity and Attestation

Ensuring the integrity of the build process and the authenticity of software artifacts is crucial. Frameworks such as `in-toto` [22] provide a systematic approach to attest to the integrity of the build process by defining a layout of steps and verifying their execution. Similarly, Sigstore's `cosign` [23] offers a widely adopted solution for signing container images and other artifacts, including SBOMs, thereby facilitating the verification of software component provenance. These tools and frameworks provide established mechanisms for artifact signing, which complements the attestation signing explored in this thesis.

1.1.4 Automated Compliance and Security Assessment Tools

Complementing SBOM generation and verification, a suite of tools has emerged to evaluate whether projects adhere to established security and conformance standards. The `OpenSSF Scorecard` [13] systematically assesses open-source software against a predefined set of security benchmarks. `Legitify` [14] performs automated evaluations to determine if a project meets specific security criteria, particularly around repository configurations. In addition, `Minder` [24] provides a platform for defining custom rules and policies on a repository, continuously monitoring compliance and taking automated actions when deviations occur. While these tools assess general security practices, a gap remains in tools specifically designed to automatically assess conformance against SSCS framework guidelines and embed these as attestations within SBOMs, which this thesis aims to address.

1.1.5 Integrated and Standardized Compliance Approaches

Recent initiatives aim to converge SBOM data with compliance verification. The Open Security Controls Assessment Language (OSCAL) [25], developed by the National Institute of Standards and Technology (NIST), provides a standardized, machine-readable format for documenting security and privacy controls. This format can complement SBOM data by offering context regarding regulatory compliance. Additionally, OWASP's authoritative guide to attestations [15] outlines the potential of incorporating compliance attestations into a CycloneDX SBOM, a concept central to our proof-of-concept. Although these approaches offer promising directions, an automated, end-to-end solution that fully integrates compliance verification *within* the SBOM generation and management lifecycle, particularly for SSCS framework guidelines, remains an open challenge.

1.2 Aim

This thesis has two linked objectives:

1. **Comparative Analysis:** To conduct a systematic, semantic comparison of five major software supply-chain security frameworks (ESF, S2C2F, SCVS, SLSA, and the SOK taxonomy) by decomposing, labeling, and semantically comparing their guidelines to measure overlap, pinpoint consensus areas, and identify gaps.

2. **Automated Attestation Proof of Concept (PoC):** To develop and evaluate a PoC tool that automatically verifies a targeted subset of these guidelines and embeds the resulting conformance attestations within an SBOM for end-to-end transparency and verifiability.

2

Background

Addressing the security challenges inherent in modern software supply chains requires structured approaches and foundational knowledge. This chapter first defines the concept of a software supply chain itself, then reviews key concepts related to SBOMs, and finally introduces prominent software supply chain security frameworks, providing the necessary context for the analysis and development presented in subsequent sections.

2.1 The Software Supply Chain

A software supply chain encompasses all the components, processes, tools, and infrastructure involved in the development, building, testing, packaging, distribution, and ongoing maintenance of a software application [26], [27]. It encompasses the software's entire lifecycle, from initial conception and coding to its eventual deployment and operation by end-users [28], [29]. Key elements typically include:

- **Source Code Development:** Writing and managing proprietary code, often using version control systems like Git.
- **Third-Party Dependencies:** Incorporating pre-existing software components, such as open-source libraries, commercial packages, and other third-party modules.
- **Build Systems and CI/CD Pipelines:** The automated processes and tools used to compile source code, run tests, package artifacts, and manage releases.
- **Artifact Repositories:** Storage systems for software artifacts, such as compiled binaries, container images, and libraries (e.g., npm, Maven Central, Docker Hub).
- **Distribution and Deployment Mechanisms:** The channels and processes through which the software is delivered to and installed by consumers or deployed into operational environments.
- **Update and Maintenance Processes:** Procedures for delivering patches, updates, and new versions of the software post-deployment.

Much like a physical manufacturing supply chain, each step and interaction within the software supply chain presents potential points of vulnerability [30], [31]. A compromise at any stage, from a malicious dependency to a compromised build server, can have cascading effects, impacting the security and integrity of the final software product and its users [32], [33]. Understanding this complex ecosystem is crucial for developing effective security measures [34]–[36].

2.2 Software Bill of Materials (SBOM)

A Software Bill of Materials is *"a formal record containing the details and supply chain relationships of various components used in building software"* [37]. SBOMs aim to enhance transparency in the software supply chain, enabling organizations

to track components, identify vulnerabilities, and manage license compliance [38]. Recognizing this, the EUs Cyber Resilience Act mandates that manufacturers of products with digital elements compile and maintain an SBOM in a commonly used, machine-readable format covering at least the top-level dependencies, to be included in their technical documentation [16].

2.2.1 Definition and Purpose of SBOMs

The primary purpose of an SBOM is to provide a detailed inventory of software components and their interdependencies. This inventory serves several key functions:

- **Vulnerability Management:** Facilitates the rapid identification of software affected by newly disclosed vulnerabilities (CVEs) by enabling cross-referencing of SBOM data with vulnerability databases.
- **License Compliance:** Aids in tracking and verifying adherence to open-source and commercial license obligations for all relevant components.
- **Supply Chain Risk Assessment:** Provides insight into the software supply chain, allowing for the evaluation of risks associated with the origin and composition of third-party and transitive dependencies.
- **Software Integrity and Origin Verification:** Establishes a detailed record, often including cryptographic hashes, that can be used to verify the integrity of installed software and trace the origin of its components.

2.2.2 Key SBOM Standards: SPDX and CycloneDX

Two widely adopted SBOM standards [38] now provide common formats for representing software component information, which enhances interoperability and supports automated processing:

- **SPDX (Software Package Data Exchange)** [39]: An open standard (ISO/IEC 5962:2021) for communicating SBOM information, including components, licenses, copyrights, security references, and relationships between them.
- **CycloneDX** [40]: A lightweight SBOM standard designed for use in application security contexts and supply chain component analysis. It supports various use cases, including vulnerability identification, license compliance, and, pertinently to this thesis, embedding attestations.

2.2.3 SBOM Generation and Tooling

SBOMs can be generated at various stages of the software development lifecycle using a variety of tools, each with unique approaches and capabilities. These tools typically analyze source code, build artifacts, or container images to identify components and their dependencies. Examples include:

- `Syft` [41]: Generates SBOMs from container images and filesystems, focusing primarily on identifying dependencies. It supports multiple output formats,

including both SPDX and CycloneDX.

- `Trivy` [42]: Primarily a vulnerability scanner, `Trivy` also offers the capability to produce SBOMs that not only list dependencies but also integrate known vulnerability information.
- `cdxgen` [43]: Dedicated to comprehensive SBOM generation in the CycloneDX format, `cdxgen` also supports populating the SBOM with software development specification data, such as from SCVS.
- `Chainloop` [44]: Provides an integrated platform that generates, stores, and visualizes SBOMs, and includes policy-as-code features for enforcing guidelines like license restrictions or version specifications.

The growing ecosystem of such tools highlight the increasing adoption and importance of SBOMs in software development and security practices.

2.2.4 SBOM Signing and Integrity Verification

The reliability of an SBOM relies on its authenticity and integrity. Cryptographic signing ensures that an SBOM has not been tampered with since its creation and verifies the identity of its issuer. The CycloneDX standard [45] itself recommends for XML and JSON-based signature schemes, such as the JSON Signature Format and JSON Web Signatures (JWS), for signing the SBOM document or specific parts thereof. This native capability is crucial for establishing trust in the attestations embedded within an SBOM, as explored in this thesis. General principles of digital signatures, involving public-key cryptography, allow any party with the corresponding public key to verify a signature made with a private key, confirming both the integrity of the data and the identity of the signer.

2.2.5 CycloneDX Attestation Model

While several SBOM formats exist, CycloneDX uniquely offers built-in support for integrating conformance information with established software guidelines. Introduced in CycloneDX version 1.6, this feature allows for the embedding of both *requirements* and *attestations* about their conformance directly within the SBOM [46]. This capability is fundamental for end-to-end compliance reporting, eliminating the need for external conventions or tools to link compliance data to the SBOM.

The CycloneDX attestation model, as illustrated in Figure 2.1, provides a hierarchical structure for representing compliance. At a high level, an *assessor* (which could be a person, an organization, or an automated tool) makes an *attestation* concerning a set of defined *requirements*. Each requirement is evaluated through one or more *claims*, which assert whether the requirement is satisfied. These claims are tied to specific *targets* (e.g., a software component, a file, or a process) and are supported by *evidence* that provides the factual basis for the claim. Should the evidence contradict a claim, *mitigation strategies* can also be documented. This structured approach allows for granular and verifiable assertions about software compliance.

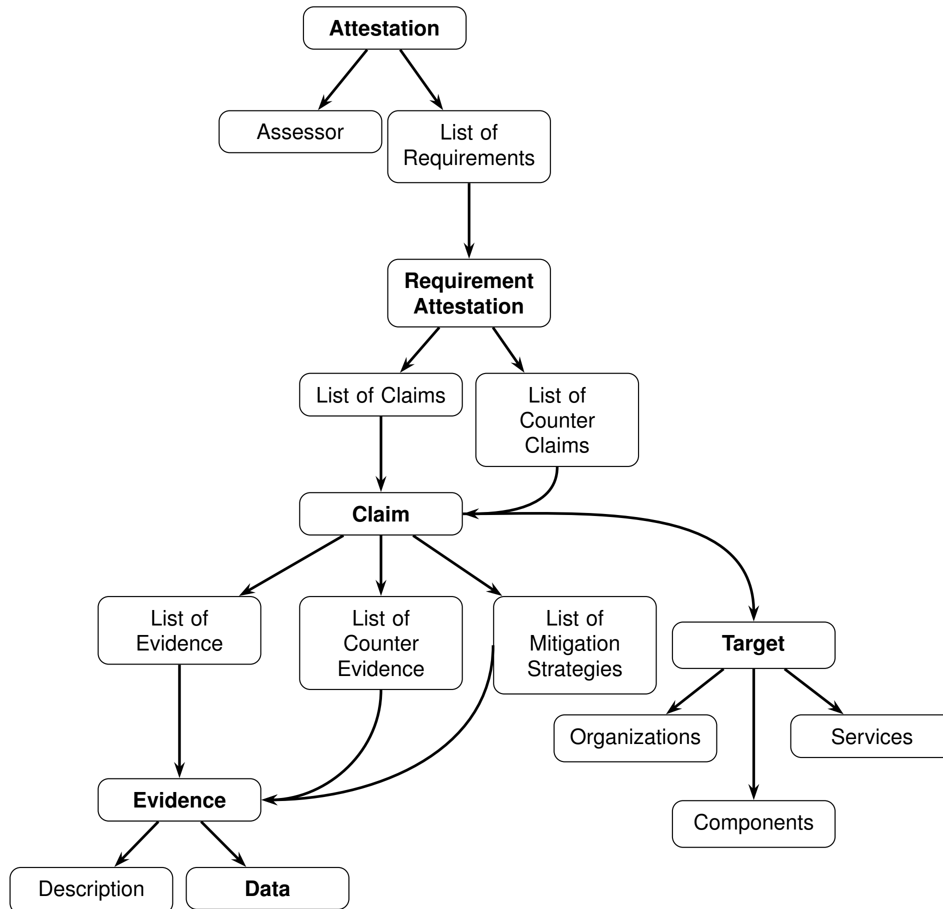

Figure 2.1: Hierarchical structure of an attestation with its supporting elements and targets.

2.3 Software Supply Chain Security Recommendations

To implement software supply chain security, several specific frameworks and sets of guidelines offer concrete recommendations for organizations and developers. These resources codify practices aimed at mitigating risks during software development, building, distribution, and consumption. This section introduces prominent examples of such recommendations, outlining their core contributions and approaches to securing the supply chain.

2.3.1 Enduring Security Framework (ESF): Recommended Practices Guide for Developers

The Enduring Security Framework (ESF) [10], a collaboration involving the National Security Agency (NSA), the Cybersecurity and Infrastructure Security Agency, the Office of the Director of National Intelligence (ODNI), and industry partners, published the *Securing the Software Supply Chain: Recommended Practices Guide for*

Developers in August 2022. This document, available to read online as a PDF, represents the first part of a series addressing different roles within the software supply chain, with this installment focusing specifically on practices for software developers.

The guide presents suggested security practices organized around the Software Development Lifecycle (SDLC). It identifies potential threat scenarios relevant to developers, such as insider threats, vulnerable dependencies, and compromises affecting the build, signing, or delivery processes. Corresponding recommended mitigations are provided for these threats, covering aspects like secure architecture, threat modeling, secure coding, testing methodologies (including static analysis, dynamic analysis, and composition analysis), and vulnerability management.

Key areas of guidance include the management of software dependencies, covering vetting processes, supplier trust, ongoing maintenance, and the use of SBOMs. The document also addresses securing the development and build environments through hardening techniques, configuration management, and secure build options. Practices related to secure code signing and the validation of final software packages before delivery are also outlined.

The ESF guide integrates with other existing frameworks. It contains an appendix mapping its recommendations to the practices defined in the NIST Secure Software Development Framework (SSDF, SP 800-218). Another appendix details the Supply-chain Levels for Software Artifacts (SLSA) framework, outlining its requirements and maturity levels related to build integrity.

2.3.2 The Secure Supply Chain Consumption Framework (S2C2F)

The Secure Supply Chain Consumption Framework (S2C2F) [9] is a specification developed under the Open Source Security Foundation (OpenSSF) to guide organizations in securely consuming open-source software components. The framework outlines a set of practices to mitigate common threats to the open-source software supply chain, including dependency confusion, malicious packages, and compromised upstream sources. It focuses specifically on the consumption phase of software supply chains, emphasizing a risk-reduction approach through structured governance and standardized workflows. The specification and its requirements are maintained in a public GitHub repository.

S2C2F is composed of eight core practices: Ingest It, Scan It, Inventory It, Update It, Audit It, Enforce It, Rebuild It, and Fix It + Upstream. These practices aim to ensure that all open-source software artifacts are consumed through controlled channels, scanned for vulnerabilities and malware, tracked in production environments, updated in a timely manner, auditable with full chain-of-custody, enforced through policy, rebuilt from trusted sources when necessary, and patched privately if required. Each practice addresses specific threat scenarios with clearly defined requirements.

The framework includes a four-level maturity model that organizes requirements into four different levels of implementation:

- **Level 1:** Basic measures such as package ingestion, scanning, and inventory management.
- **Level 2:** Focuses on reducing response time to vulnerabilities and improving ingestion configuration.
- **Level 3:** Includes proactive threat mitigation, malware scanning, and enforcement of curated feeds.
- **Level 4:** Involves high-assurance activities such as rebuilding artifacts from source and submitting private fixes upstream.

S2C2F is solution-agnostic and also provides implementation guidance and tooling examples for each requirement. The documentation also includes threat mappings, practical use cases, assessment guidelines, and reference architectures.

2.3.3 Software Component Verification Standard (SCVS)

The OWASP Software Component Verification Standard (SCVS) [7] is a framework developed to address software supply chain security by providing a structured set of controls and best practices on an interactive website. It aims to mitigate risks associated with the integration of third-party and open-source components by establishing guidelines that cover various aspects of software production. The framework is organized into six groups: Inventory, Software Bill of Materials, Build Environment, Package Management, Component Analysis, and Pedigree & Provenance.

The guidelines within SCVS specify a range of activities and verification controls that are designed to be implemented incrementally through a layered maturity model. This model defines three levels of assurance:

- **Level 1:** Addresses basic best practices applicable to any software system.
- **Level 2:** Suitable for organizations with established risk management frameworks.
- **Level 3:** Intended for environments that require high assurance due to the sensitivity of the data or the critical nature of the systems involved.

This tiered approach allows organizations to assess their current security posture and gradually enhance their supply chain assurance measures over time.

2.3.4 Supply-chain Levels for Software Artifacts (SLSA)

Supply-chain Levels for Software Artifacts (SLSA) [8] is a security framework consisting of a set of standards and guidelines designed to improve the integrity of software artefacts and mitigate supply chain threats. Developed through industry consensus and currently managed by the Open Source Security Foundation (OpenSSF), SLSA aims to provide a common language and structure for discussing and enhancing software supply chain security practices. The framework is motivated by the need to protect against tampering at various software lifecycle stages, from source code creation to building, packaging, and distribution. Similar to SCVS, the framework

and supporting documentation can be viewed and explored online on a dedicated website.

SLSA is organized into levels and tracks. Levels (currently 0-3 in v1.0) represent incrementally increasing security assurances for a given aspect of the supply chain. Tracks allow for focusing on specific security aspects independently. SLSA version 1.0 primarily defines the requirements for the *Build Track*, which concentrates on the integrity of the build process itself and the link between the source code and the resulting artefact. Future versions are anticipated to define additional tracks, potentially covering source code management and dependency handling in more detail.

A core component of SLSA is *provenance*, which is authenticated metadata describing how a software artifact was produced. SLSA specifies a recommended format for this provenance, based on the in-toto attestation framework. This provenance typically includes information about the build platform, the inputs to the build (source code references, dependencies, build parameters), and the identity of the entity performing the build. The framework emphasizes the principle of trusting build platforms and verifying the artifacts they produce, rather than implicitly trusting artifacts based on configuration alone. It also promotes tracing software back to verifiable source code and favors explicit, signed attestations as evidence over inferred properties.

The SLSA Build Track levels provide progressive guarantees:

- **Level 1** requires that build provenance exists and is distributed to consumers, primarily aiding in documentation and understanding the build process.
- **Level 2** adds the requirement that the provenance is generated by a hosted build service and is authenticated (typically via digital signature), protecting against provenance tampering after the build.
- **Level 3** further requires the build platform itself to be hardened, ensuring strong isolation between build instances and protecting the provenance generation process from interference by the build steps themselves.

SLSA distinguishes itself from, yet complements, concepts like SBOMs. While SBOMs typically list the components within a software artifact, SLSA provenance focuses on certifying the integrity of the build process. SLSA is intended for use by software producers to improve their practices, software consumers to evaluate the trustworthiness of artifacts, and infrastructure providers to offer secure build and distribution services. Verification involves checking an artifact's provenance against expectations defined for that specific software component, confirming builder identity, source origins, and build parameters.

2.3.5 Systematization of Knowledge (SOK): Taxonomy of Attacks on Open-Source Software Supply Chains

Ladisa et al. (2023) present a Systematization of Knowledge (SOK) paper that provides a comprehensive taxonomy of attacks targeting open-source software sup-

ply chains [11]. The work aims to create a common reference and terminology for understanding these threats, independent of specific programming languages or ecosystems. A comprehensive, interactive list of identified attack vectors and corresponding mitigations can also be viewed[1].

The core contribution is a detailed taxonomy structured as an attack tree, identifying 107 unique attack vectors covering the supply chain from code contribution to final package distribution. This taxonomy was developed through a systematic literature review encompassing both scientific and grey literature, including real-world incident reports. It categorizes attacks based on the stage of the supply chain targeted (source, build, distribution) and the degree of interference with legitimate project artifacts and processes. Examples of high-level attack categories include creating name confusion with legitimate packages, subverting legitimate packages by injecting malicious code into sources, tampering during the build process, or compromising the distribution of packages.

Complementary to the attack taxonomy, the authors identify and classify 33 general safeguards intended to mitigate the described attack vectors. These safeguards represent the guideline content of the work and are categorized based on control type (e.g., preventive, detective) and the stakeholders involved in their implementation (e.g., project maintainers, administrators, downstream consumers). Examples include maintaining detailed SBOMs, implementing secure authentication practices like MFA, conducting code reviews, using code signing, employing secure build environments, utilizing vulnerability scanning, and establishing vetting processes for dependencies.

The research methodology included validation of both the attack taxonomy and the identified safeguards through two user surveys: one with 17 domain experts and another with 134 software developers. These surveys assessed the taxonomy's correctness, comprehensiveness, and comprehensibility, as well as the perceived utility and cost of the safeguards. The results also provide insights into developer awareness and adoption levels for various security practices.

[1]Risk Explorer for Software Supply Chains: Interactive attack-tree visualization

3

Methods

This chapter describes the research methodology employed in this thesis. The approach first examines existing software supply chain security frameworks and then explores their automated application.

Section 3.1 details a systematic comparative analysis of selected software supply chain security frameworks. This involves framework selection, guideline decomposition, semantic similarity assessment, and thematic labeling to understand the scope, overlap, and gaps in current guidance. The goal is to provide a structured overview of the theoretical landscape.

Section 3.2 describes the design and implementation of a PoC for automated guideline conformance. This includes methods for embedding verifiable compliance attestations within SBOMs, developing automated checkers, cryptographically signing attestations, and creating a tool for visualizing and verifying these enriched SBOMs. This practical work aims to demonstrate the feasibility of applying software supply chain security principles.

3.1 Comparative Analysis

This section outlines the specific steps taken for the comparative analysis of the selected software supply chain security frameworks, as illustrated in Figure 3.1. To achieve this evaluation, the analysis proceeds through several stages, described in the following subsections: the rationale and process for framework selection (3.1.1), an initial examination of guideline representation (3.1.2), the method for decomposing guidelines into granular units (3.1.3), the technique for measuring semantic similarity between these units (3.1.4), and the approach for labeling decomposed guidelines to analyze thematic coverage (3.1.5).

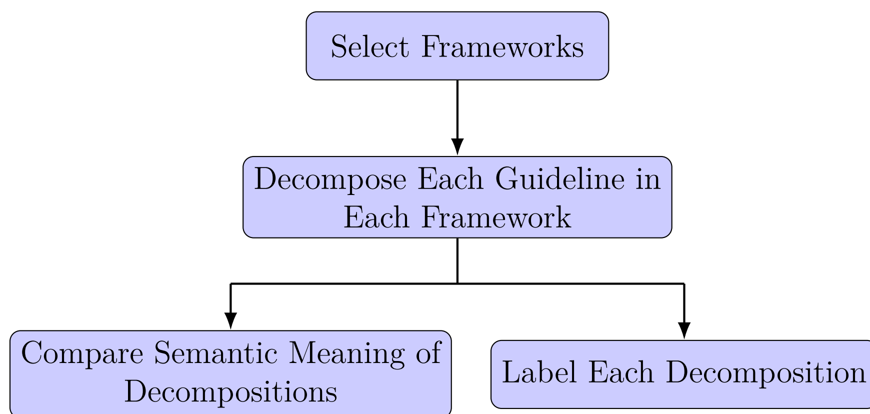

Figure 3.1: Overview of the steps taken in the comparative analysis.

3.1.1 Framework Selection

A wide range of theoretical frameworks exist to provide recommendations and guidelines that developers can adopt to enhance software security [7]–[11]. These frameworks address various aspects of software development and supply chain security,

including secure coding practices, dependency management, and risk mitigation strategies. However, since this thesis specifically focuses on software supply chain security, only frameworks with a dedicated emphasis on this domain have been considered.

Several key criteria were applied to select the frameworks for this study:

1. The selected frameworks had to focus explicitly on *software supply chain security* rather than general software development practices.

2. The selection aimed to incorporate perspectives from both *industry* and *academia* to ensure a well-rounded analysis. Industry-driven frameworks may provide practical, widely adopted security measures, while academic contributions offer structured classifications and theoretical insights that support a deeper understanding of software supply-chain threats.

3. An effort was made to include frameworks that align with *government regulations and official security guidelines*, reflecting the increasing role of public policy in securing software supply chains in response to high-profile attacks and national cybersecurity initiatives.

Based on these criteria, five frameworks were identified that collectively represent a diverse set of practices and methodologies for securing the software supply chain:

- **ESF - Securing the Software Supply Chain: Recommended Practices for Developers**: Provides comprehensive recommendations for developers to enhance software supply chain security. The framework is aligned with regulatory efforts and emphasizes secure coding, hardened build environments, and dependency verification [10].

- **S2C2F - Secure Supply Chain Consumption Framework**: Offers guidelines for securely consuming open-source software within supply chains. It establishes best practices for evaluating the security of external dependencies and mitigating risks associated with third-party components [9].

- **SCVS - OWASP Software Component Verification Standard**: SCVS is a community-developed framework that outlines key activities, security controls, and best practices to assess and minimize risks in the software supply chain [7].

- **SLSA - Supply-chain Levels for Software Artifacts**: Defines a security framework aimed at improving the integrity of software build artifacts [8].

- **SOK: Taxonomy of Attacks on Open-Source Software Supply Chains**: Presents a systematic classification of software supply chain attacks. This taxonomy provides insights into various attack vectors and lists a set of safeguards associated with these attack vectors [11].

3.1.2 Framework Analysis and Guideline Representation

The selected frameworks were examined to identify structural and conceptual differences in how they define and present their security guidelines. A key challenge encountered during this analysis was the considerable variation in how guidelines were formulated and structured across different frameworks.

One of the primary differences observed was in the length and granularity of individual guidelines. Some frameworks provided concise, checklist-style recommendations, while others included more detailed descriptions that integrated background information and justifications. Another notable difference was the way recommendations were phrased. Certain frameworks used permissive language, such as *"Project Maintainer can establish a patch management process..."* [11], while others employed more prescriptive phrasing, for instance *"Scan OSS for malware"* [9]. In some cases, frameworks also included declarative statements alongside recommendations, for example: *"A Software Bill of Materials (SBOM) is a list of components... Project Maintainers can produce and distribute SBOMs..."* [11].

Additionally, the formatting and organization of guidelines varied significantly. Frameworks such as SCVS structured their recommendations in a clearly defined list format, making individual guidelines easy to identify. In contrast, frameworks like ESF presented guidelines in extensive text-based reports, often embedded within paragraphs of explanatory content. These differences in structure made direct comparisons between frameworks challenging, as aligning guidelines based purely on their descriptions required subjective interpretation.

3.1.3 Decomposition of Guidelines

We aimed to compare the aspects of each framework that can be interpreted as guidelines, particularly those that enable consumers to take action. However, this was difficult due to the differences we identified in Section 3.1.2 above. Thus, to enable a structured comparison between security guidelines from different frameworks, we implemented a two-step process:

1. **Extraction of Guidelines** Identifying and standardizing the original guidelines.
2. **Decomposition of Guidelines** Breaking down each guideline into smaller, independent statements.

3.1.3.1 Extraction of Guidelines

Guidelines were first extracted from each framework using methods tailored to their individual formats:

ESF: The ESF guidelines appear within a comprehensive report, with security recommendations embedded in explanatory text. To standardize extraction, only the recommendations explicitly listed in the "Recommended Mitigations" sections were taken. This ensured that all extracted content represented actionable security measures rather than general background information.

S2C2F: In the S2C2F framework, recommendations are organized within a structured requirements table. Each requirement title listed in the “Secure Supply Chain Consumption Framework Requirements” table was extracted as an independent guideline to ensure consistency.

SCVS: The SCVS framework presents its recommendations as a structured list. Every guideline was taken directly as listed, ensuring that each control was included in the analysis without modification.

SoK: Unlike SCVS and S2C2F, the SOK framework offers a taxonomy of attacks and associated safeguards. Each safeguard was extracted as an individual guideline, representing explicit security measures that can be compared to those in other frameworks.

SLSA: The SLSA framework organizes its recommendations into progressive levels for supply chain security. Individual guidelines were extracted from the Producing artifacts and Distributing provenance sections of the SLSA v1.0 specification. Additional recommendations that were tightly coupled with the SLSA build system were excluded due to their limited general applicability.

In total, 284 guidelines were extracted (see Table 4.1).

3.1.3.2 Decomposition Process

As some guidelines contained compound or multi-component recommendations, we decomposed each guideline into smaller, granular statements—referred to as *decompositions*. Formally, let

$$\mathcal{G} = \{g_1, g_2, \ldots, g_N\}$$

be the set of extracted guidelines. For each guideline $g \in \mathcal{G}$, we define its decomposition as a set of statements:

$$\mathcal{D}(g) = \{d_1, d_2, \ldots, d_m\} \quad (m \geq 1)$$

such that the following conditions hold:

1. **Actionable:** Each decomposed statement should be actionable; therefore, stand-alone factual statements were removed.
2. **Preservation of Intent:** The combined semantic content of the decomposed statements is equivalent to that of the original guideline’s actionable content, i.e.,
$$\text{Meaning}(g) = \text{Combine}(d_1, d_2, \ldots, d_m).$$
3. **Independence:** Each decomposed statement d_i is self-contained and understandable without additional context.
4. **Explicitness:** Only information explicitly stated in g is included in $\mathcal{D}(g)$; no implicit assumptions are added.
5. **Standardization of Action Verbs:** Action verbs (e.g., “can”, “should”) are normalized to ensure a consistent interpretation across all statements.

In practice, this process involved:

- Splitting compound or multi-component guidelines into individual, stand-alone statements.
- Ensuring that each decomposed statement retains the original guideline's meaning and purpose.

The following example illustrates the decomposition process:

Original Guideline:

"Use linters and static analysis tools."

Decomposed Statements:

- Use linters.
- Use static analysis tools.

While prior work on automated policy extraction, such as Text2Policy [19] and recent LLM-based event-extraction models [20], has demonstrated decent accuracy for extracting policies from text, we considered their performance inadequate for our highly specialized security guidelines. The work by Wanner *et al.* [21] further highlights the possibility to automatically decompose a statement into its atomic subclaims but deems that future work is still needed for a more reliable solution. Additionally, no research to date has evaluated these methods on our specific dataset of supply-chain security recommendations, meaning we could not predict their reliability, coverage, or error characteristics in our context. Furthermore, due to our approach of further decomposing the guidelines themselves, rather than merely extracting actionable statements, these approaches may not be applicable. We therefore chose a manual decomposition process rather than relying on automation, whose real-world efficacy had not been verified.

Overall, this decomposition process yielded 1321 decomposed statements from the original 284 guidelines, as seen in Table 4.1.

3.1.4 Semantic Comparison of Decomposed Guidelines

A semantic similarity analysis was conducted on the decomposed guidelines to systematically compare the security guidelines across different frameworks. Given the structural and linguistic differences in how each framework formulates its security recommendations, relying on simple lexical overlap (e.g., keyword matching) or basic textual comparison methods would be insufficient. Two guidelines can convey the same core meaning despite differing in wording, sentence structure, and specific terminology used. Such surface-level differences could lead to many false negatives (missing true similarities) if only exact textual matches were considered.

Furthermore, due to the large number of decompositions (1321 statements), manual comparison becomes impractical, excessively time-consuming, and introduces a high degree of human bias and subjectivity. Therefore, an automated, reproducible, and semantically-aware approach was preferred.

While recent studies suggest that LLMs can perform well in evaluating nuanced semantic similarity [47], their direct application to the extensive pairwise comparisons required in this thesis was considered less practical. This is primarily due to the heavy processing demands and the time required to process large numbers of sentence pairs. Consequently, an automated and reproducible method based on sentence embeddings was selected, as it offers greater efficiency and scalability for large-scale comparative analysis.

Several non-LLM-based approaches exist for measuring semantic similarity, with deep neural network-based methods currently demonstrating the strongest performance [48]. Among these, the MPNet model proposed by Kaitao et al. [49] has shown the highest effectiveness.

`all-mpnet-base-v2`, a pre-trained transformer-based language model, was employed for this analysis. This model, a fine-tuned variant of MPNet, is presently regarded as the best-performing sentence transformer [50]. By utilizing this model, a comprehensive semantic similarity analysis was made possible, allowing for the identification of conceptually equivalent guidelines across frameworks, even when their textual expressions differed.

Each decomposed guideline was first encoded into an embedding, which was then stored and cached to optimize computational efficiency. To compare decompositions, the default metric, cosine-similarity was used as the primary metric, measuring the angular distance between embeddings in vector space. A similarity score between 0 and 1 was assigned to each guideline pair, where 1 indicates near-identical meaning and 0 indicates no semantic relationship.

Each framework's decomposed guidelines were compared against those of the other frameworks to identify semantically similar guidelines. Guidelines from the same framework were not evaluated against each other. Instead, comparisons were only made between decomposed guidelines from different frameworks. This ensured that the analysis focused on cross-framework similarities rather than redundancies within individual frameworks.

3.1.5 Labeling of Decomposed Guidelines

While the semantic similarity analysis reveals granular overlaps between individual guidelines, it does not, by itself, offer a high-level thematic perspective on the scope, emphasis, or potential coverage gaps within each framework. To obtain these broader insights and support a more structured comparison of the frameworks content, we adopted an inductive qualitative content analysis approach inspired by Mayrings model [51]. This categorization allowed us to identify the predominant security domains addressed by each framework, comparing their thematic distributions, and detecting areas of convergence or distinctive specialization.

Our labeling process, therefore, consisted of two main stages:

3.1.5.1 Initial Category Formation

We began by forming an initial label set $\mathcal{L}$ derived from our selected frameworks' categories, including, for example, build pipeline security, dependency management, governance, and policy. Although this initial set was designed to capture the most prevalent security aspects, we anticipated that it might require refinement when applied to the diverse range of decompositions.

3.1.5.2 Iterative Sampling and Category Refinement

Following Mayrings constant comparison logic [51], for each framework F with decompositions $\mathcal{D}_F$, we repeated the following until we deemed our categories were sufficient for allowing a detailed thematic comparison:

1. **Sampling:** Randomly select a subset $S_F \subseteq \mathcal{D}_F$ of 30 decompositions.
2. **Coding:** Map each $d \in S_F$ to one or more categories in $\mathcal{L}$.
3. **Evaluation:** Check that every $d \in S_F$ is covered by at least one category and that all categories in $\mathcal{L}$ are used at least once.
4. **Revision:** If any d falls outside $\mathcal{L}$ or some categories remain unused, refine definitions, merge or split categories, or add new ones.
5. **Resampling:** Apply the revised $\mathcal{L}$ to a fresh S_F until no further adjustments are needed.

3.1.5.3 Final Application to the Full Corpus

Once the category system stabilized, we applied the final mapping function

$$f : \mathcal{D} \to \mathcal{P}(\mathcal{L})$$

to the complete set of decompositions $\mathcal{D}$. Ideally, each $d \in \mathcal{D}$ satisfies

$$f(d) \subseteq \mathcal{L} \quad \text{and} \quad f(d) \neq \emptyset.$$

However, despite our iterative refinement, a small number of decompositions could not be unambiguously mapped to any label. These decompositions were kept unlabeled.

3.2 Automated SBOM-Based Compliance Attestation

Building upon the insights derived from the comparative analysis of theoretical frameworks (Section 3.1), this section details the methodology developed for a PoC system aimed at automating software supply chain security compliance verification. The primary objective is to establish and demonstrate a practical, end-to-end process for assessing adherence to these guidelines, embedding these assessments as verifiable **attestations** within SBOMs, and enabling accessible inspection and validation. A high-level overview of the system is seen in Figure 3.2 below.

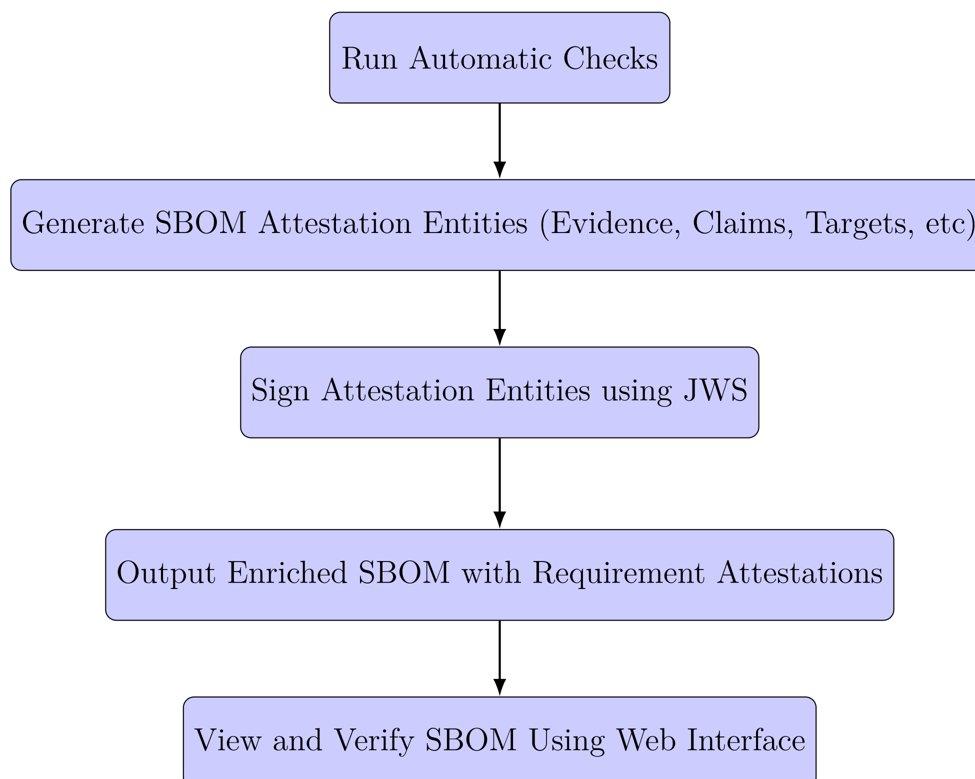

Figure 3.2: High-level diagram of the PoC system.

At the heart of this PoC is the concept of an **attestation**: a verifiable assertion about a software project's conformance to specific security guidelines. We leverage the CycloneDX SBOM format, which natively supports embedding such attestations (the model itself is detailed in Section 2.2.5), allowing for a self-contained approach to compliance reporting.

The subsequent subsections will progressively build our PoC methodology:

- We begin by detailing how compliance information is structured as attestations within SBOMs using a specific interpretation of the CycloneDX model suitable for automation (Section 3.2.1).
- Next, we explain how the decomposed guidelines (from Section 3.1) serve

as the foundation for creating targeted automated conformance checks (Section 3.2.2).

- The overall workflow for the automated generation and embedding of these compliance attestations is then mapped out (Section 3.2.5).
- We then delve into the design and implementation of the specific compliance checkers developed for this PoC, including how conformance and confidence scores are derived from their outputs (Section 3.2.3).
- To ensure the trustworthiness of the generated attestations, we present our cryptographic signing strategy using JSON Web Signatures, supported by a defined security threat model (Section 3.2.4).
- To facilitate human review and verification, the design and functional requirements for a dedicated SBOM attestation viewer and verifier tool are outlined (Section 3.2.6).
- Finally, we describe the experimental setup for a feasibility study designed to evaluate this end-to-end PoC system (Section 3.2.7).

3.2.1 Structuring Conformance Evidence within SBOMs for the PoC System

Our PoC system leverages the CycloneDX attestation model to embed conformance evidence directly within SBOMs. As detailed in Section 2.2.5, which also illustrates the general structure in Figure 2.1, CycloneDX was chosen for its native support for both *requirements* definitions and *attestations* regarding their conformance [46]. This enables a self-contained approach to compliance reporting. In our case, the decomposed guidelines described in Section 3.1.3 are treated as requirements.

The hierarchical organization presented in the CycloneDX model (Figure 2.1) is adopted by our PoC. In our system, an *assessor* (the PoC tool itself) generates an attestation regarding a set of *requirements*. Each requirement's conformance is detailed through *claims*, stating whether the requirement is met for a given *target* (e.g., a specific software project). These claims are supported or refuted by *evidence* derived from automated checks.

While the CycloneDX specification provides a rich and flexible schema with numerous optional fields, our PoC system focuses on a selected subset of these fields. This selection was driven by the need to identify the minimal yet sufficient data required for trustworthy automated requirement assessment. This section details these core entities and the specific fields we have employed within our automated compliance tool.

3.2.1.1 Requirement

This data object describes a specific requirement that could, for example, originate from a security framework, a compliance standard, or an organizational policy. In

our case, it encapsulates the details of a decomposed guideline that software or processes are expected to adhere to.

Fields in the Requirement Object

- `bom-ref`: This field serves as a unique identifier for the requirement entry within the current data structure or SBOM. It can be used for internal linking or referencing. In the provided example, it mirrors the framework's identifier (`"ING-1"`).
- `identifier`: This field holds the requirement's own specific ID as defined by the framework or standard it originates from. This allows for easy cross-referencing with the source documentation of the requirement (e.g., `"ING-1"`).
- `parent`: The `parent` field indicates a higher-level grouping, category, or module to which this specific requirement belongs. This helps in organizing requirements hierarchically. For instance, `"P1-Ingest_It"` suggests it's part of a broader category named "P1-Ingest_It".
- `title`: This field provides a concise, human-readable title or summary of the requirement. It gives a quick understanding of the requirement's objective.
- `text`: The `text` field offers a more detailed description or rationale for the requirement. It could elaborate on the title or explain the benefit or necessity of adhering to it.

JSON Example of a Requirement

```
{
    "bom-ref": "ING-1",
    "identifier": "ING-1",
    "parent": "P1-Ingest_It",
    "title": "Use public package managers trusted by your organization (e.g., NuGet.org, npmjs.com,
    ↪  PyPi.org, etc.)",
    "text": "Your organization benefits from the inherent security provided by the package manager"
},
```

3.2.1.2 Target

This data object describes a target, which in the context of CycloneDX can be an organization, a service, or a component. However, for this PoC, only the component type is used, as a component represents a discrete software or hardware element, which is what the checks consider.

Fields in the Target Object

- `bom-ref`: This attribute serves as a unique identifier for this specific target (component) within the SBOM itself. This reference allows other parts of the SBOM to link back to this target unambiguously. The example directly references the exact version of the GitHub repository react using the commit hash, which enables traceability for the consumer who is inspecting the SBOM.

- `name`: This field provides a human-readable name for the component that can be easily understood.
- `type`: This attribute categorizes the component. The CycloneDX specification details a comprehensive list of valid types, including `library`, `framework`, `application`, `container`, `platform`, `operating-system`, `device`, `firmware`, and `file`. The `application` type can be used to describe a software project or repository, which is what many of the checks target.
- `description`: This optional field offers a more detailed, human-readable explanation of the component and can provide more context than the `name`.
- `hashes`: This is an array containing one or more cryptographic hashes of the component. Each object within this array specifies the hashing `alg` (algorithm, e.g., `SHA-256`) and the actual hash `content` (the hash value). This enables SBOM consumers to verify the integrity of a component, ensuring that the examined target is indeed the target that was processed.

JSON Example of a Target

```
{
  "bom-ref": "https://github.com/facebook/react/tree/6060367ef8a7a5bac12e0f830367bb13626db83a",
  "name": "GitHub repository 'facebook/react'",
  "type": "application",
  "description": "The GitHub repository 'facebook/react' at commit
  ↪  6060367ef8a7a5bac12e0f830367bb13626db83a. Available at:
  ↪  https://github.com/facebook/react/tree/6060367ef8a7a5bac12e0f830367bb13626db83a",
  "hashes": [
    {
      "alg": "SHA-256",
      "content": "ce71576997c7f6e2774875a499a85e8..."
    }
  ],
  "signature": {
    "algorithm": "RS256",
    "value": "eyJhbGciOiJSUzI1NiJ9.eyJib20tcmVmIj..."
  }
}
```

3.2.1.3 Assessor

The `assessor` data object is used to name the entity responsible for specific evaluations or for the generation of parts of the SBOM itself. This entity could be an automated software tool, a third-party auditing firm, or an internal security team. Documenting the assessor provides a clear point of reference for any claims or findings reported in the SBOM. As this was a PoC, the specific data was not of great importance or relevant to highlighting the trustworthiness of the conformance to guidelines in the SBOM. Thus, the example is very minimal.

Fields in the Assessor Object

- `bom-ref`: This field is the unique identifier for the assessor object within the SBOM document. This allows other sections of the SBOM, such as an

attestation, to reference this specific assessor by its `bom-ref`.

- `organization`: This field is an object that provides details about the organization to which the assessor belongs or represents.
  - `name`: A sub-field within `organization`, this specifies the human-readable name of the assessing organization. In the illustrative example, this is `"SBOM Tool"`. The CycloneDX specification allows for other details within the `organization` object, such as contact information, although they are not present in this specific minimal example.

JSON Example of an Assessor

```json
{
    "bom-ref": "sbom-tool",
    "organization": {
      "name": "SBOM Tool"
    }
}
```

3.2.1.4 Evidence (E)

An `evidence` object serves as a record of a specific check or analysis that was performed. It captures the output or findings of a check, providing a verifiable data point. In this example, it contains the results of an automated check that verifies the integration of vulnerability scanners into a CI/CD pipeline.

Fields in the Evidence Object

- `bom-ref`: This field acts as a unique identifier for this specific piece of evidence within the SBOM. This allows the evidence to be referenced from other parts of the document, for example, within a `claim` that relies on this evidence.
- `propertyName`: This field specifies the particular property, check, or control that this evidence pertains to. It gives context to what was being assessed. For example, `"cicd.vulnerabilityScanners"` indicates that the evidence relates to the presence or configuration of vulnerability scanners within the continuous integration and continuous deployment pipeline.
- `description`: This field provides a human-readable summary of the evidence or the check that was performed. It explains the purpose or outcome in a high-level manner.
- `data`: This field is an array that holds the actual detailed findings of the check. Each item in this array can represent a distinct piece of data or output.
  - `name`: Within each data item, this gives a specific title to this piece of evidence data.
  - `contents`: This object encapsulates the actual evidence.

    * `attachment`: This object has a `content` field which holds the raw evidence data. This content is often a stringified JSON object containing detailed results, messages, and confidence scores from the check, as seen in the example: `"{"name":"VulnerabilityScannerCheck", "passed":false,"message":..."}`.
    * `contentType`: This specifies the MIME type of the `content`, such as `"application/json"`, informing how the content should be interpreted.

- `created`: This timestamp shows when the evidence was generated or when the check was performed.
- `expires`: An optional timestamp that can be included. This suggests a point in time after which the evidence might be considered stale or no longer valid, prompting a reassessment.
- `author`: This object provides information about the entity (person or tool) that generated this evidence.
  - `name`: The name of the authoring tool or person.
  - `email`: An optional contact email.

JSON Example of an Evidence

```
{
  "bom-ref": "evidence-cicd.vulnerabilityscanners-vulnerabilityscannerchecker-2837b429",
  "propertyName": "cicd.vulnerabilityScanners",
  "description": "Automatic check for vulnerability scanner integration in CI/CD pipelines has been
  ↪  run",
  "data": [
    {
      "name": "VulnerabilityScannerCheck",
      "contents": {
        "attachment": {
          "content": "{\"name\":\"VulnerabilityScannerCheck\",\"passed\":false,\"message\":\"No
          ↪  vulnerability scanner integration found in GitHub Actions
          ↪  workflows\",\"detail\":\"GitHub Actions workflows were found, but no security scanning
          ↪  keywords detected.\",\"confidence\":0.7,\"confidenceRationale\":\"GitHub Actions
          ↪  workflows exist but no security-related keywords found.\",\"weight\":1}",
          "contentType": "application/json"
        }
      }
    }
  ],
  "created": "2025-05-19T09:43:14.874Z",
  "expires": "2026-05-19T09:43:14.874Z",
  "author": {
    "name": "SBOM Tool",
    "email": "sbom-tool@example.com"
  },
  "signature": {
    "algorithm": "RS256",
    "value": "eyJhbGciOiJSUzI1NiJ9.eyJib20tcmVmIjoiZXZpZGVuY2..."
  }
},
```

3.2.1.5 Counter-Evidence (CE)

Evidence that contradicts or weakens a Claim. The data elements for Counter-Evidence are identical in structure to *Evidence*, but are interpreted as negative support against a Claim.

Claim (C)

This data object represents a signed assertion or statement about whether something is true that is used to convince the consumer that a `Requirement` is being followed. It links evidence, reasoning, and potential mitigation strategies to this assertion as a way to prove to the consumer that the statement is correct.

Fields in the Claim Object

- `bom-ref`: This field serves as a unique identifier for this specific claim within the SBOM or data structure. It allows the claim to be referenced by the `Attestation` object, defined below.
- `target`: This field identifies the specific target to which this claim applies. It is defined as a pointing to a target object defined elsewhere in the SBOM.
- `predicate`: This field contains the core assertion or statement of the claim. It's a human-readable description of what is being claimed about the target.
- `counterEvidence`: This array can list or reference pieces of evidence that might contradict or challenge the main evidence supporting the claim.
- `evidence`: This is an array of identifiers (`bom-refs`) that point to `Evidence` objects defined elsewhere. This links the claim to the concrete data or observations that support it.

JSON Example of a Claim

```
{
  "bom-ref": "claim-no-vulnerability-scanner-9080c7c9",
  "target": "https://github.com/facebook/react/tree/6060367ef8a7a5bac12e0f830367bb13626db83a",
  "predicate": "No vulnerability scanner integration was found in CI/CD pipelines for checking
↪  vulnerable components",
  "counterEvidence": [],
  "evidence": [
    "evidence-cicd.vulnerabilityscanners-vulnerabilityscannerchecker-2837b429"
  ],
  "signature": {
    "algorithm": "RS256",
    "value": "eyJhbGciOiJSUzI1NiJ9.eyJib20tcmVmIjoiY2xhaW0tbm8tdnVs..."
  }
},
```

3.2.1.6 Counterclaim

A Claim whose predicate asserts non-conformance. In the attestation map it appears under `counterClaims`. The data elements for a Counterclaim are identical in structure to a *Claim*, but with a negated predicate (e.g., "Linters are NOT used").

Attestation (A)

This data object represents an aggregation of all claims and counterclaims related to a set of requirements. It provides an overall assessment, performed by a specific assessor, summarizing the conformance and confidence levels for each evaluated requirement based on the collected claims.

Fields in the Attestation Object

- `summary`: A human-readable summary or title for the attestation, providing context for the overall assessment. For example, `"Automatic assessment using SBOM Tool"`.
- `assessor`: A `bom-ref` that points to the `assessor` object responsible for performing this attestation.
- `map`: This is an array of objects, where each object represents the assessment details for a specific requirement.
  - `claims`: An array of identifiers `bom-refs`) pointing to `claim` objects that support the conformance to the requirement.
  - `counterClaims`: An array of identifiers (`bom-refs`) pointing to `claim` objects that act as counterclaims, indicating non-conformance or disputing positive claims for the requirement.
  - `confidence`: An object detailing how confident we are that the conformance of the requirement is correct.
    - `score`: A numerical value between 0 and 1 representing the confidence level of how confident we are that the `conformance` is correct.
    - `rationale`: A textual explanation of how the confidence score was derived.
  - `conformance`: An object detailing the conformance status for this requirement.
    - `score`: A numerical value between 0 and 1 representing the conformance level (e.g., `0`, indicating non-conformance in the example).
    - `rationale`: A textual explanation of how the conformance score was determined.
  - `requirement`: An identifier for the specific requirement being assessed in this map entry (e.g., `"SG-0.36"`). This corresponds to the `bom-ref` from a requirement defined in the SBOM.

JSON Example of an Attestation


```
{
  "summary": "Automatic assessment using SBOM Tool",
  "assessor": "sbom-tool",
  "map": [          {
      "claims": [],
      "confidence": {
        "score": 0.7,
        "rationale": "Overall confidence in this assessment is the weighted average of all
        ↪  contributing evidence confidences. Total Weight: 1.0. Evidence for claim
        ↪  claim-no-vulnerability-scanner-9080c7c9 (isCounter: true) contributed with:
        ↪  [Confidence: 0.70, Weight: 1.0 (Rationale: GitHub Actions workflows exist but no
        ↪  security-related keywords found.)]."
      },
      "conformance": {
        "rationale": "Conformance score is the ratio of positive claims to total claims.",
        "score": 0
      },
      "requirement": "SG-0.36",
      "counterClaims": [
        "claim-no-vulnerability-scanner-9080c7c9"
      ]
    }],
    "signature": {
        "algorithm": "RS256",
        "value": "eyJhbGciOiJSUzI1NiJ9.eyJzdW1tYXJ5IjoiQXV0b21hdGljI..."
  }
}
```


3.2.2 Leveraging Decomposed Guidelines for Automated Checks

The automated requirement conformance checks developed in this study operate on the decomposed guidelines, the derivation of which is detailed in Section 3.1.3. This selection of granular, atomic statements as the requirements for automated verification, rather than the original, often multifaceted, framework guidelines, influences the design and interpretation of the conformance checks. We chose this approach due to a number of reasons:

Firstly, decomposed statements offer improved clarity and reduced ambiguity for the logic of automated checkers. Original guidelines can be extensive, potentially combining multiple distinct requirements. Decomposed statements, by design, represent singular and focused conditions. This characteristic is particularly relevant when translating requirements into logic for automated systems, as it enables each check to be mapped to a distinct, atomic condition. Secondly, atomic statements are inherently more easily tested. It is considerably simpler to design, implement, and validate an automated check for a single, clearly defined condition (e.g., "*Use linters.*") compared to architecting a single, complex check for a compound guideline (e.g., "*Use linters and static analysis tools, and ensure they are integrated into the CI/CD pipeline with blocking capabilities for critical findings.*"). The latter would necessitate intricate logic within a solitary check, thereby increasing its complexity and the likelihood of errors or incomplete verification.

Furthermore, due to conformance being tied to the requirement, it means that when an automated check operating on a decomposed statement fails, the specific reason

for non-conformance is immediately apparent; the precise atomic requirement was not fulfilled. In contrast, if a check designed against a complex original guideline indicates a failure, further manual investigation is frequently required to determine which specific clause or component of the guideline was violated. Thus, employing decomposed statements facilitates more precise, actionable feedback from the automated system, aiding in quicker remediation.

3.2.3 Automatic Compliance Checks

To demonstrate the versatility of our automated compliance tool and its capability to address diverse aspects of the software supply chain, this section details our PoC, which implements automated checks for a variety of guideline categories. We first define what an automated checker entails in our system and the nature of its output. Subsequently, we outline the rationale for selecting specific guidelines for this PoC and how they map to checker implementations. We then describe the methodology for calculating conformance and confidence scores from checker outputs, followed by detailed descriptions of the implemented PoC checkers.

In our system, an automated "checker" is an individual, programmatic routine designed to verify a specific aspect of a software component or process against a defined criterion. One or more such checkers assess a single compliance guideline, each focusing on a distinct, verifiable part of that guideline.

Upon execution, each checker instance yields key information about its findings, including:

- **A unique identifier** for the specific check performed (this is the class name of the checker).
- **A clear pass or fail status** indicating the outcome of the verification.
- **A human-readable message** summarizing the result.
- **Specific details or context** about the findings, which also identify the precise artifact or configuration assessed (the "target").
- **A confidence score** (on a scale of 0.0 to 1.0) reflecting the certainty in the reported status. This is particularly important for checks that rely on heuristics. A CI pipeline checker identifying build commands in workflow files illustrates this; the confidence for a positive result is not absolute, as these commands can exist in non-executed contexts, such as comments, within workflow files.
- **An explanation for the assigned confidence level**.
- **A machine-readable code** categorizing the type of failure, if applicable.
- **A relative weight** indicating the check's significance (for confidence aggregation).

The compliance tool processes structured information from individual checkers to add detailed guideline conformance data to an SBOM. An initial SBOM is required for this; the tool can either use a pre-existing one or, if not provided, generate a new CycloneDX-compliant SBOM using `cdxgen` [43].

This addition of conformance data is achieved by methodically transforming checker outputs into specific CycloneDX entities. The “pass or fail status” directly deter-

mines whether a `Claim` (asserting satisfaction of a guideline) or a `Counterclaim` (asserting non-satisfaction) is generated for the guideline. The "specific details or context" and "human-readable message" become `Evidence` objects supporting the assertion. The checker's findings identify the `Target` (the artifact or configuration assessed), while its "confidence score" and "explanation" inform the confidence level of the generated `Claim`/`Counterclaim` and its `Evidence`.

3.2.3.1 Targeted Guidelines and Checker Mapping for PoC

For our PoC, we aimed to select a representative subset of guidelines to demonstrate the application of our proposed attestation mechanisms across various aspects of software supply chain security. The specific guidelines targeted for automated checker development were chosen based on two primary considerations:

1. **Feasibility for a Proof of Concept:** We prioritized guidelines that were considered relatively straightforward to implement as automated checkers. This approach allowed a focus on demonstrating the core attestation system, rather than on automating more complex or context-dependent guidelines.
2. **Demonstration of Broad Applicability:** To ensure the PoC addressed different security topics, we selected guidelines corresponding to distinct labels derived from our framework comparative analysis (see Section 3.1.5 for the labeling process and Table 4.3 for the complete set of labels).

Table 3.1 details the specific guidelines selected based on these criteria, along with their assigned labels.

Table 3.1: Guidelines targeted by PoC and their assigned labels

Guideline Text	Framework	ID	Assigned Labels
Each SBOM has a unique identifier.	SCVS	2.3	• Software Bill of Materials (SBOM)
Application uses a continuous integration build pipeline.	SCVS	3.3	• Build Process
Securely configure your package source files (i.e. nuget.config, .npmrc, pip.conf, pom.xml, etc.).	S2C2F	ENF-1	• Dependency & Package Consumption • Secure Development Environment

Continued on next page

Table 3.1: Guidelines targeted by PoC and their assigned labels (continued)

Guideline Text	Framework	ID	Assigned Labels
Open Source Consumers can disable script execution during package installation.	SOK	SG-010	• Dependency & Package Consumption
Project Maintainers can enable protection rules for sensitive branches.	SOK	SG-016	• Development Process Controls
Open-source consumers can integrate vulnerability scanners into their CI/CD pipelines to check for vulnerable components.	SOK	SG-036	• QA, Testing & Security Scanning

The specific checkers developed to verify these selected guidelines are mapped in Table 3.2. This table serves as an overview of the implementation details discussed in the subsequent section.

Table 3.2: Mapping of guidelines to implemented PoC checkers

Framework	ID	Implemented PoC Checker Class(es)
SCVS	2.3	`CycloneSerialNumberPresenceChecker`, `CycloneUUIDFormatChecker`
SCVS	3.3	`ContinuousIntegrationPipelineChecker`
S2C2F	ENF-1	`PackageSourceChecker`
SOK	SG-010	`ScriptExecutionChecker`
SOK	SG-016	`BranchProtectionChecker`
SOK	SG-036	`VulnerabilityScannerChecker`

3.2.3.2 Calculation of Conformance and Confidence Scores

For each guideline targeted by the PoC (as detailed in Table 3.1), the structured outputs from its associated automated checker(s) (mapped in Table 3.2) are aggregated and evaluated to produce an overall conformance score and a confidence score for the assessment.

Before detailing the calculation methods, it is important to note the rationale behind their selection. For this PoC, our primary goals were clarity and ease of implementation, enabling us to clearly demonstrate how individual checker outputs are combined into meaningful compliance and confidence scores. The chosen formulas, while simple, effectively demonstrate this aggregation.

3.2.3.2.1 Conformance Score The conformance score for a guideline is determined by the proportion of positive assertions of compliance (Claims) relative to the total number of assertions (both Claims and Counterclaims) made by the checkers for that guideline. Calculating the score as a direct ratio of positive outcomes to total checks provides a straightforward measure of adherence, clearly indicating the extent to which a guideline's requirements, as evaluated by the checkers, have been met. It is calculated as:

$$\text{Conformance Score} = \frac{\text{Number of Positive Assertions (Claims)}}{\text{Total Number of Assertions (Claims + Counter-Claims)}} \tag{3.1}$$

By applying this formula, a single check yields a definitive outcome—either a pass or a fail—for the respective requirement, as the result is determined solely by one assertion. However, when multiple checkers are involved, the score may indicate partial compliance. For instance, consider the guideline *"Each SBOM has a unique identifier"* (SCVS 2.3): if one checker confirms the presence of an identifier, while another reports uncertainty regarding its uniqueness, the resulting conformance score would be 50%. This proportion appropriately reflects that the requirement is only partially fulfilled.

3.2.3.2.2 Confidence Score The overall confidence in the assessment of a guideline is derived from the confidence levels associated with the evidence provided by each checker. The calculation process is as follows:

1. Each piece of evidence gathered by a checker contributes its own confidence score and an assigned weight.
2. All individual evidence confidence scores from all checkers for a specific guideline are collected, along with their respective weights and textual rationales.
3. The final confidence score is calculated using a weighted average of all contributing evidence confidences. This method is used because individual pieces of evidence can differ in their reliability or relevance. By applying weights, evidence considered more certain or important is given greater influence on the guideline's overall confidence score:

$$\text{Final Confidence Score} = \frac{\sum(\text{Evidence Confidence}_i \times \text{Evidence Weight}_i)}{\sum \text{Evidence Weight}_i} \tag{3.2}$$

A detailed textual rationale accompanies the final confidence score. This rationale summarizes how the overall confidence was derived, including the total weight of evidence considered, and includes a log detailing the contribution of each piece of evidence (its specific confidence, weight, and individual rationale). This detailed logging provides transparency into the aggregated confidence value.

3.2.3.3 Check Implementation Details

The following subsections detail the implementation approach for each specific checker class listed in Table 3.2. It is important to acknowledge that many compliance guidelines can be open to interpretation. The checker implementations described herein

represent our specific understanding and approach to verifying these guidelines for this PoC.

While these checkers offer one method for automated assessment, they are not exhaustive, and other valid approaches or more comprehensive checks could certainly be developed. As this work is a PoC, our primary focus has been on demonstrating the general feasibility of automatically checking compliance for a diverse set of guidelines, rather than implementing comprehensive checker logic.

For this PoC, the checker implementations concentrate on Node.js based projects within the context of repositories hosted on GitHub. Consequently, some checks are platform-specific, such as those interacting with the GitHub API, while others are more general, like those analyzing SBOMs, or focus on project-local configurations.

`BranchProtectionChecker`

This check verifies if sensitive branches in a GitHub repository have protection rules enabled. The implementation interacts with the GitHub API's branch listing endpoint (e.g., `/repos/{owner}/{repo}/branches`) to retrieve information for all branches. It then filters for the predefined sensitive branches: `main`, `master`, and `develop`. For each of these sensitive branches found, it examines the `protected` status returned by the API to determine if protection rules are active. The check assesses if all, some, or none of the identified sensitive branches are protected.

`ContinuousIntegrationPipelineChecker`

This check inspects the project repository for evidence of a Continuous Integration (CI) pipeline, specifically verifying GitHub Actions. The verification process involves locating the `.github/workflows` directory and parsing any YAML workflow files within it. The content of these files is analyzed to identify jobs (by their ID or `name` attribute) or specific steps (by their `run` command). It looks for keywords like "build" in job names or identifiers, and for common build commands (matched via a regular expression including terms like `npm run build`, `mvn package`, `gradle build`, `dotnet build`, `cargo build`, `make`, or generic terms like "compile") within step commands. The presence of such a job or step suggests an active CI build pipeline.

`CycloneSerialNumberPresenceChecker`

This checker operates directly on the input CycloneDX SBOM. It verifies that the `serialNumber` field exists at the root level of the SBOM document and contains a non-empty value. The check passes if the field is present and populated with any string. This, in conjunction with `CycloneUUIDFormatChecker`, helps satisfy SCVS 2.3.

`CycloneUUIDFormatChecker`

This checker operates on the input CycloneDX SBOM. If a `serialNumber` is present (as verified by `CycloneSerialNumberPresenceChecker`), this check validates its

format. According to the CycloneDX specification, the serial number must be an RFC 4122 URN UUID. This checker specifically verifies the `urn:uuid:` prefix and then uses a regular expression to ensure the subsequent string matches a standard UUID version 4 pattern.

`PackageSourceChecker`

This checker aims to mitigate dependency confusion attacks by verifying key aspects of a project's package management setup. It specifically checks for:

- Secure package registry URLs.
- The presence of appropriate package manager lock files.
- Pinned (exact) dependency versions in the project's manifest.

To verify secure package sources, the checker examines the project's manifest file (specifically its publishing configuration) and relevant package manager configuration files. It ensures that any configured package registry URLs use the secure HTTPS protocol or match a predefined list of known secure registries.

Additionally, it confirms the presence of the project's package manager lock file.

Finally, the checker assesses dependency versions declared in the project's manifest file to ensure they are pinned to exact versions. Versions are identified as unpinned if they use range specifiers (e.g., `^`, `~`, `>`, `<`, `||`, ` - `) or wildcards (e.g., `*`, `x`, `X`).

`ScriptExecutionChecker`

This check verifies if the project provides configuration that allows consumers to disable the execution of package installation scripts. The implementation inspects specific project-local configuration files for common Node.js package managers. For npm and pnpm, it checks `.npmrc` files (in the project root or package manager specific subdirectories) for the setting `ignore-scripts=true` or simply `ignore-scripts`. For Yarn (v2+), it checks `.yarnrc.yml` files (in project root or `.yarn/` subdirectory) for the setting `enableScripts: false`. The presence of these exact configurations indicates that script execution can be controlled.

`VulnerabilityScannerChecker`

This check determines if a project has integrated vulnerability scanning for its dependencies within a CI/CD pipeline. The verification process inspects GitHub Actions YAML workflow files located within the `.github/workflows` directory. These files are parsed to search their content for keywords or command patterns using regular expressions. One regular expression targets names of known vulnerability scanning tools or services (e.g., `trivy`, `snyk`, `npm audit`, `codeql`). Another targets more general security-related terms (e.g., "security scan", "vulnerability check", "SAST"). Detection of matches for these patterns suggests the integration of vulnerability scanning.

3.2.4 Signing Attestations for Integrity and Authenticity

To ensure the trustworthiness and verifiability of the requirement conformance data embedded within the SBOM, a robust signing mechanism is essential. Digital signatures provide a cryptographic means to guarantee the **integrity** and **authenticity** of the attested information. Integrity ensures that the signed data has not been altered since it was signed, while authenticity verifies the identity of the signer. This is typically achieved using public-key cryptography, where an entity signs data with their private key, and any party with access to the corresponding public key can verify the signature.

The primary accomplishment of this signing method is to create a verifiable chain of trust from the raw evidence to the final attestation. It allows consumers of the SBOM to confirm that a claimed assessor indeed issued the reported compliance status and its supporting data. Additionally, it ensures that the data has not been tampered with.

However, it is important to discuss what digital signing does not inherently accomplish. Signing a piece of data does not, by itself, guarantee the *correctness* or *accuracy* of the information being signed. For instance, if an automated checker has a flaw in its logic and produces incorrect evidence, signing that evidence merely authenticates that the flawed evidence originated from the checker, not that the evidence itself is true. Similarly, it does not protect the private keys themselves from compromise; secure key management is a prerequisite. While there are various methods to ensure data integrity, such as checksums or HMACs, they do not provide the same level of authenticity and non-repudiation offered by digital signatures based on asymmetric cryptography.

3.2.4.1 Threat Model

This threat model outlines the security context for the digital signing strategy applied to SBOM attestations. It defines the assets being protected, the assumed adversary profile, their objectives, and key operational assumptions. The primary purpose is to understand the protections afforded by the implemented signing mechanisms.

Assets at Risk
The core assets this signing strategy aims to protect are:

- **Integrity of Attestation Data:** Ensuring that compliance attestations, individual claims, supporting evidence, and target specifications within the SBOM have not been altered after they are signed.
- **Authenticity of Attestation Source:** Verifying that attestations originate from the claimed assessor(s) and not an impersonator.

Adversary Profile
We consider an adversary with the following general characteristics and capabilities:

- **Knowledge and Access:**

  - Capable of obtaining and reading any SBOM, as these documents are often shared across organizations or may be publicly accessible.
  - Able to intercept and inspect data in transit or at rest if not otherwise protected (e.g., by transport-layer security).
- **Technical Capabilities:**
  - Can attempt to modify or inject data into SBOMs.
  - Can analyze the structure, format, and content of legitimate attestations from various sources to understand their construction.
  - Can attempt to craft data that mimics the appearance of legitimate attestations or assessor identities.

Adversary Goals
Given the adversary profile, their primary goals concerning the signed attestations include:

- **Data Tampering:** To illicitly modify signed compliance attestations, claims, evidence, or targets within an SBOM without detection. The intent could be to:
  - Falsely represent adherence to a security guideline (e.g., changing a non-conformant status to conformant).
  - Falsely represent non-adherence to a security guideline (e.g., to discredit a product or supplier).
- **Attestation Forgery/Impersonation:** To create and introduce new, entirely forged attestations into an SBOM that appear legitimate, or to alter existing attestations to appear as if they originated from a different, trusted assessor.

Key Assumptions and Scope Limitations
The effectiveness of the signing strategy against the described threats relies on the following critical assumptions:

- **Private Key Secrecy:** The adversary **does not** have access to the legitimate private keys used by assessors for signing. The compromise of a private key is outside the direct protection scope of the signature verification mechanism itself and would invalidate its guarantees.
- **Cryptographic Strength:** The cryptographic algorithms and parameters used for generating and verifying signatures (e.g., JWS with RS256) are assumed to be computationally secure against known cryptanalytic attacks within the relevant timeframe.
- **Focus:** This threat model primarily addresses threats to the integrity and authenticity of the attestation data *after* it has been generated and signed. It does not explicitly cover:
  - Vulnerabilities within the automated checkers or systems that generate the pre-signed data (i.e., ensuring the accuracy of the information before it is signed).
  - The operational security practices for managing private keys by legitimate assessors.

3.2.4.2 Implementation of Signing using JSON Web Signatures

For the practical implementation of signing within our CycloneDX SBOMs, we have chosen the **JSON Web Signature (JWS)** format, as specified in RFC 7515. JWS is a standard method for representing signed content using JSON data structures and is natively supported in the CycloneDX specification. We used the RS256 algorithm, which is an RSA-based signature algorithm and therefore utilizes public-key encryption.

In our system, digital signatures are applied at multiple levels within the attestation structure to defend against unauthorized modifications. Specifically, each piece of `Evidence`, each `Claim` (and `Counterclaim`), each `Target` reference, and ultimately the encompassing `Attestation` object can be individually signed. This multi-level signing ensures that an adversary lacking the private keys cannot alter these individual components such as the content of evidence, the predicate of a claim, or the identifiers of a target without invalidating the JWS signature. Forging a new valid signature for any maliciously altered data is cryptographically infeasible.

This approach establishes a verifiable chain of trust from the raw, signed `Evidence` (guaranteeing its integrity since creation by an automated checker or manual input) to a signed `Claim`. The claim's signature binds this verified evidence to a specific assertion about a requirement and, more importantly, to a specific, signed `Target`. This explicit, signed linkage to a target, defined by unique identifiers such as hashes or repository URLs, is crucial. It prevents an adversary from deceptively reusing a valid, signed claim or attestation from one project (Project A) to falsely assert compliance for an unrelated project (Project B), since the target would reference Project A and not B. Any attempt to alter the target information within a signed claim to point to Project B would invalidate the claim's signature. Thus, even with access to numerous SBOMs, an attacker cannot easily abuse signed attestations across different contexts if the SBOM consumers diligently verify that the target information corresponds to the project under evaluation.

Finally, the signature on the overall `Attestation` object authenticates the aggregated assessment, including conformance and confidence scores, as originating from the assessor and protects it from modification. This makes it significantly harder for an adversary to undetectably modify any part of the compliance narrative, such as changing a conformance score or selectively omitting claims, as the signature on the encompassing attestation would be broken.

A noteworthy advantage of this granular signing approach is its support for multiple assessors using different key pairs to sign different components. This is particularly useful where, for example, an automated tool signs its findings, while a third-party auditor signs their manual verification of a distinct requirement such as *"Engineers within the development organization should be required to take annual training of organization-approved cybersecurity best practices."*, all within the same SBOM. This distributed trust model, where different entities vouch for specific aspects, is fully verifiable through their respective signatures.

Furthermore, in addition to the granular signing of individual attestation compo-

nents, we also support signing the entire SBOM document itself using JWS. This provides a comprehensive integrity and authenticity check for the SBOM as a whole, ensuring that the complete set of components, metadata, and all embedded attestations have not been tampered with or selectively omitted since the SBOM was finalized and signed by its author or publisher.

While this signing architecture significantly raises the bar for attackers and provides strong assurances of data integrity and authenticity, the ultimate trustworthiness also relies on the security of the private keys and the diligence of SBOM consumers. Verifiers must not only validate signatures and trust the assessor but also confirm the contextual relevance of the signed data, especially the specified targets. Consumers must confirm that the targets and evidence referenced are actually part of the project to which the SBOM refers. The system, however, provides the necessary means to enable this type of verification.

3.2.5 Execution Flow

Our automated compliance tool follows this general execution flow:

1. **Ingest Requirements:** Populate the CycloneDX SBOM with *Requirement* definitions, (e.g. from SCVS, S2C2F).
2. **Generate Evidence:** Run automated checks to emit *Evidence* (E) and *Counter-Evidence* (CE) objects.
3. **Form Claims:** For each Requirement, group the relevant E and CE into *Claims* (C) or *Counterclaims.*
4. **Assemble Attestations:** For each Requirement, collect all associated Claims and Counterclaims into an *Attestation* (A) object, computing overall conformance and confidence metrics for each requirement.
5. **Sign and Embed:** Digitally sign all generated Evidence, Claims, Counterclaims, and the final Attestation. Embed these signed objects into the SBOM.

3.2.6 Design of an SBOM Attestation Viewer and Verifier

A dedicated tool was developed for the visualization and cryptographic verification of compliance attestations embedded within SBOMs. The primary objective of this tool is to translate machine-readable compliance data from SBOMs into a format that is both readily understandable and verifiably trustworthy for human review, thereby improving transparency and confidence in conformance. The tool was conceptualized as a web-based application for accessibility and ease of use.

The following key functional requirements (FR) were defined for the tool:

1. **FR1: SBOM Ingestion and Compliant Parsing.** The tool must be capable of ingesting SBOM files in the CycloneDX JSON format. It must accurately parse the `declarations` section of the SBOM, which includes attestation elements (such as `Requirements`, `Attestations`, `Claims`, `Evidence`, and `Targets`) and requirement specifications.

2. **FR2: Hierarchical Display of Compliance Data.** The tool must present

compliance information in a clear, hierarchical structure. This structure should allow users to navigate from an overview of assessed frameworks and standards down to individual requirements and their specific attestations, facilitating easier identification and examination of relevant compliance details.

3. **FR3: Detailed Attestation Component Inspection.** Users must be able to drill down into each attestation to view its constituent parts. This includes associated claims and counter-claims, the specific targets (e.g., software components, files) to which these claims apply, and the detailed supporting evidence. This functionality is essential for auditability and understanding the basis of each compliance assertion.

4. **FR4: Digital Signature Verification and Status Indication.** The tool must provide a mechanism for users to input public keys corresponding to assessors. Subsequently, it must perform cryptographic verification of all embedded digital signatures (JWS) within the attestations, claims, evidence, and targets. The verification status (e.g., valid, invalid, missing signature) for each signed element must be clearly and unambiguously displayed to the user. This allows the consumers to easily identify which attestations have not been modified since the related trusted party signed them.

The implemented PoC tool was designed to meet these functional requirements. It accepts CycloneDX JSON files via a standard file selection interface (addressing FR1). Upon loading an SBOM, the tool presents a comprehensive overview of the attested frameworks and their requirements, as illustrated in Figure 3.3, fulfilling FR2.

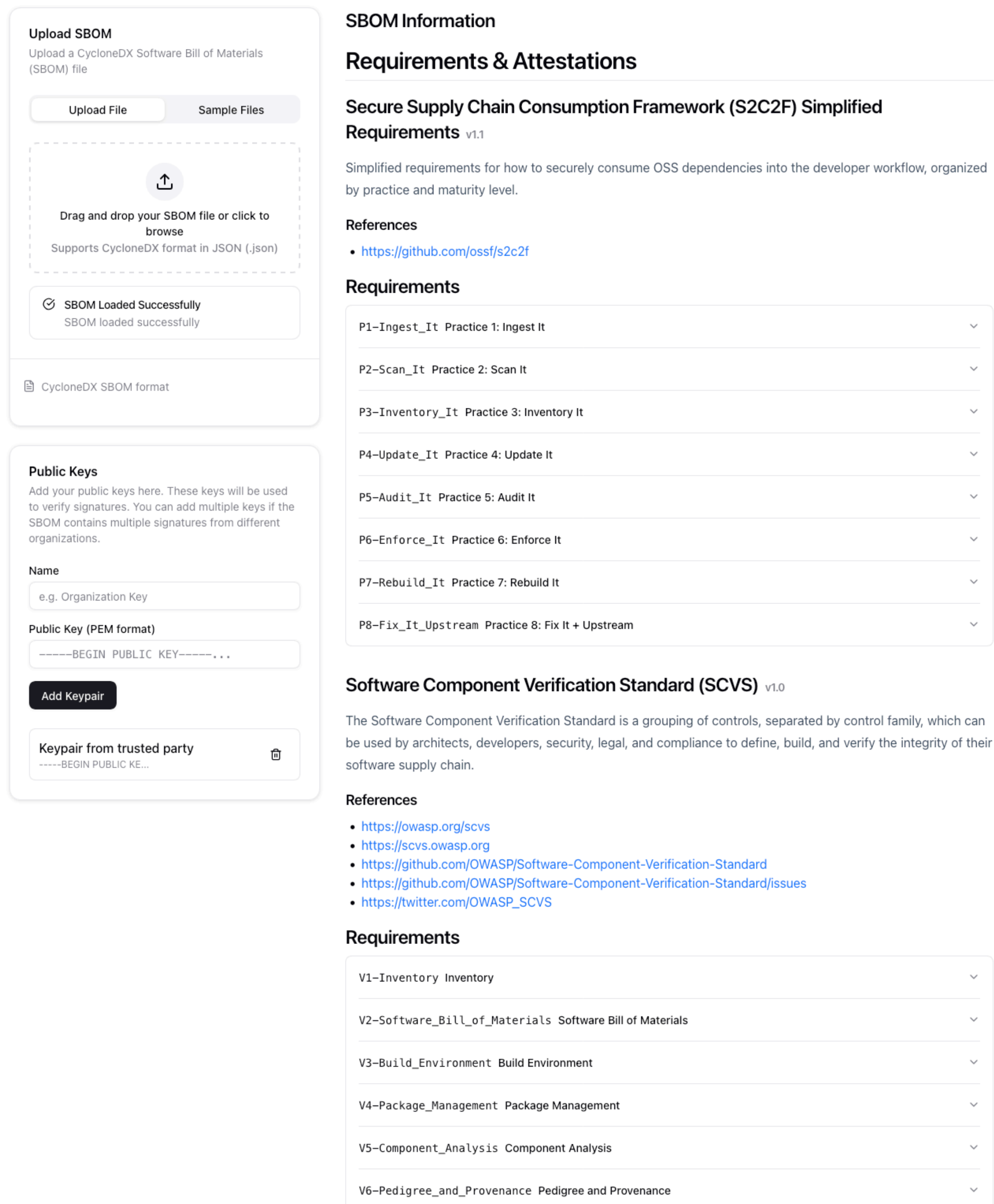

Figure 3.3: Example overview display in the visualization tool (addresses FR2).

To address FR3, the tool allows users to expand individual requirements to inspect detailed claims and the evidence supporting them (Figure 3.4). Furthermore, details of specific targets referenced in claims can be viewed (Figure 3.5).

2.3 Each SBOM has a unique identifier 100% 100% 2 Claims

Each SBOM has a unique identifier SBOM Tool Self

Conformance

100%

Rationale: Conformance score is the ratio of positive claims to total claims.

Confidence

100%

Rationale: Overall confidence in this assessment is the weighted average of all contributing evidence confidences. Total Weight: 2.0. Evidence for claim claim-there-exists-a-bbff8c0e (isCounter: false) contributed with: [Confidence: 1.00, Weight: 1.0 (Rationale: BOM serialNumber field is present.)]. Evidence for claim claim-the-serial-number-69e43123 (isCounter: false) contributed with: [Confidence: 1.00, Weight: 1.0 (Rationale: Serial number is a valid URN:UUID v4.)].

Claims

There exists a serial number in the SBOM fastify.json Comp

The serial number is in the proper URN:UUID format fastify.json Comp

Evidence

Automatic check of the serial number is in the proper URN:UUID format has been run

Metadata

ID: `evidence-bom.serialnumber-cycloneuuidformatchecker-cc224342`

Property: `bom.serialNumber`

Created: 2025-05-19, 23:42:32 Expires: 2026-05-19, 23:42:32

Author

SBOM Tool

sbom-tool@example.com

Data

UUIDFormatCheck

application/json


```
{
  "name": "UUIDFormatCheck",
  "passed": true,
  "message": "UUID format check passed. Valid URN:UUID format:
urn:uuid:4471962d-3960-49d7-bd71-a1ec71b83308",
  "detail": "The serial number is properly formatted as a URN:UUID.",
  "confidence": 1,
  "confidenceRationale": "Serial number is a valid URN:UUID v4.",
  "weight": 1
}
```


Figure 3.4: Example detailed expanded view for an SCVS requirement (addresses FR3).

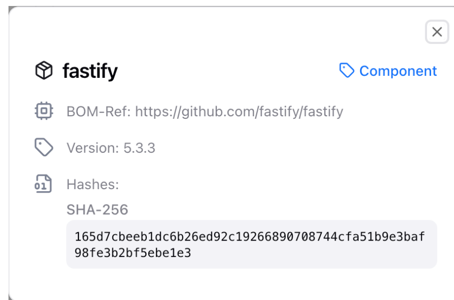

Figure 3.5: Example modal window displaying target component details (addresses FR3).

For FR4, the tool includes a mechanism for users to input and manage public keys of trusted assessors as seen in Figure 3.3. It then performs cryptographic verification of JWS. The outcomes are clearly visualized: Figure 3.6 shows an example of a tool dialog confirming successfully verified signatures, while Figure 3.7 illustrates how the tool indicates issues if signatures are invalid, missing, or only partially verifiable. This verification status is also integrated directly into the requirement listing for immediate user feedback, as shown in Figure 3.8.

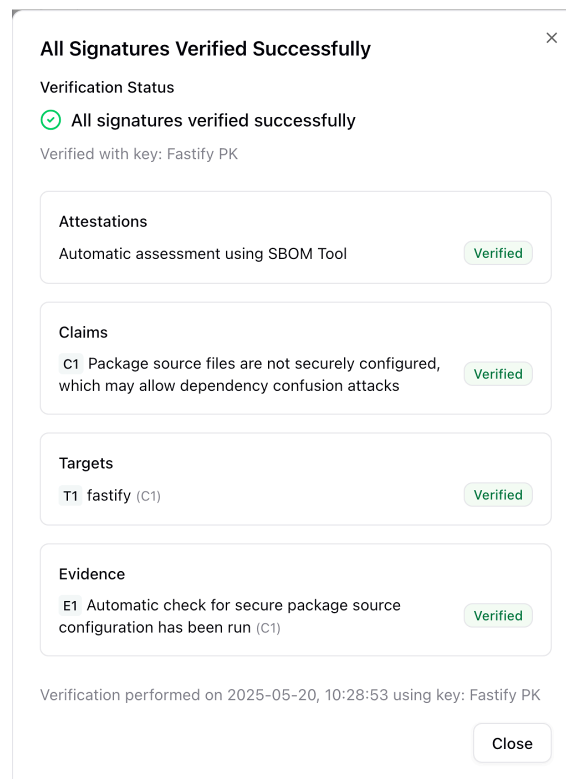

Figure 3.6: Example tool dialog confirming successful signature verification (addresses FR4).

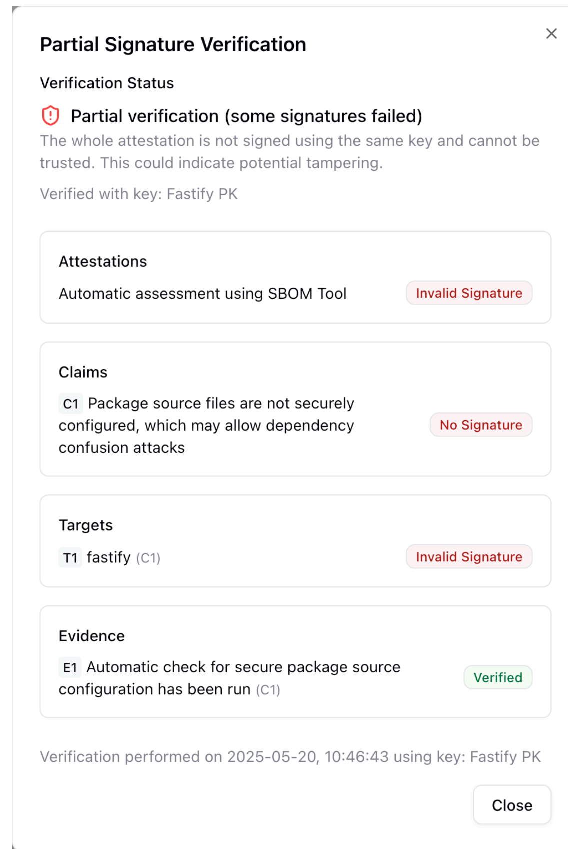

Figure 3.7: Example tool dialog indicating issues with signature verification (addresses FR4).

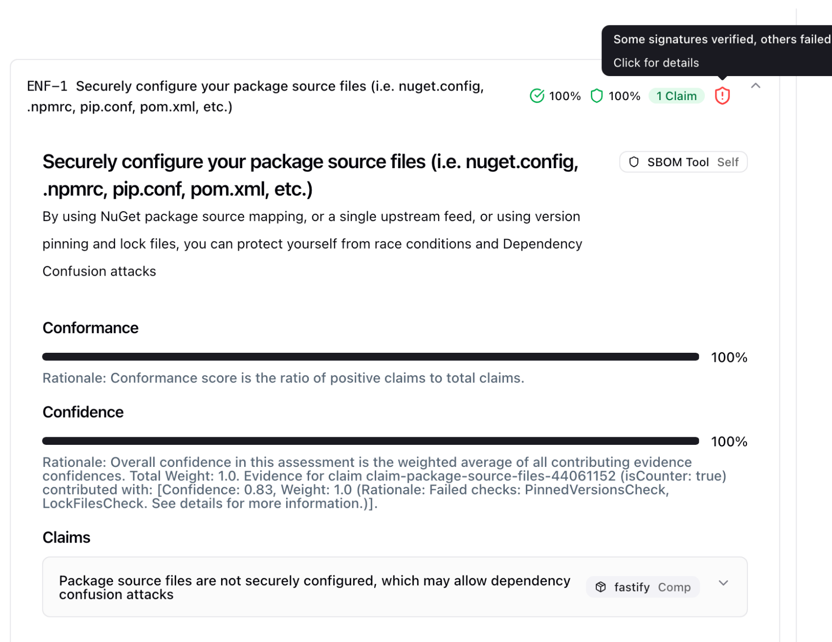

Figure 3.8: Example of integrated signature verification status in the requirement view (addresses FR4).

3.2.7 Feasibility Study Experimental Setup

To demonstrate the practical feasibility of the end-to-end automated compliance checking and attestation process outlined in this thesis, a feasibility study was conducted. The primary objectives of this study were to:

1. **Validate the End-to-End Workflow:** Demonstrate that the proposed workflow—from ingesting requirements, running automated checks, generating structured attestations (Evidence, Claims, Targets), embedding these within a CycloneDX SBOM, cryptographically signing them, to finally visualizing and verifying the enriched SBOM—is operationally feasible.
2. **Illustrate Automated Checking for Varied Guidelines:** Show that the system can accommodate checks for different types of software supply chain security guidelines, ranging from SBOM-specific requirements to repository configurations and CI/CD practices, even with PoC level checkers.
3. **Demonstrate Verifiability:** Confirm that the generated attestations, once signed and embedded, can be effectively parsed, displayed, and their cryptographic signatures verified by a consumer, thereby establishing trust in the attested information.

The focus of this study is on the process viability and the functionality of the attestation framework, rather than the comprehensiveness or definitive accuracy of the individual PoC checkers themselves. The checkers serve as a means to generate data for the attestation pipeline.

3.2.7.1 Target Projects

Five widely-adopted open-source projects from the Node.js ecosystem were selected for this study. These projects are:

1. Express.js (`https://github.com/expressjs/express`): A minimal and flexible Node.js web application framework.
2. Axios (`https://github.com/axios/axios`): A promise-based HTTP client for the browser and Node.js.
3. Fastify (`https://github.com/fastify/fastify`): A fast and low overhead web framework, for Node.js.
4. NestJS (`https://github.com/nestjs/nest`): A Node.js framework for building efficient, reliable and scalable server-side applications.
5. React (`https://github.com/facebook/react`): A JavaScript library for building user interfaces.

3.2.7.2 Selected Guidelines for Automation

The specific guidelines against which compliance was checked in this feasibility study are those previously detailed in Table 3.1. The PoC automated checkers corresponding to these guidelines (mapped in Table 3.2) were utilized to generate the necessary

inputs for demonstrating the end-to-end attestation generation, signing, and verification workflow. This selection allowed for testing the system across various security aspects and technical domains pertinent to software supply chain security.

4

Results

4.1 Comparative Analysis

This section details the comparative analysis of five key software supply chain security frameworks: ESF, S2C2F, SCVS, SLSA, and the SOK taxonomy. Following the methodology outlined in Section 3.1, which involved guideline decomposition, semantic similarity assessment, and thematic labeling, we investigate the scope, content, and interrelation of their recommendations. The subsequent subsections present quantitative and qualitative findings on guideline structure (Section 4.1.1), semantic overlap (Section 4.1.2), thematic distribution (Section 4.1.3), and an overall framework coverage and suitability analysis (Section 4.1.4). These results aim to clarify the current software supply chain security guidance landscape.

4.1.1 Decomposition Results

In this section, the quantitative results of decomposing guidelines from various software supply chain frameworks are presented. The decomposition process involved splitting compound guidelines (e.g., *"Use linters and static analysis tools"*) into individual statements (e.g., *"Use linters"* and *"Use static analysis tools"*).

4.1.1.1 Decompositions per Framework

Table 4.1 summarizes the number of guidelines and their corresponding decompositions across the five frameworks, with a total of 284 guidelines decomposed into, 1321 individual statements.

Table 4.1: Summary of the number of guidelines and their decompositions for each software supply chain framework.

Framework	Guidelines	Decompositions
ESF	119	822
S2C2F	25	49
SCVS	81	137
SLSA	16	82
SOK	43	231
Total	**284**	**1321**

Figure 4.1 presents the number of decompositions per guideline. The average number of decompositions ranges from approximately 1.7 to nearly 7 across the frameworks, with some guidelines having as few as one and others reaching up to 36 decompositions.

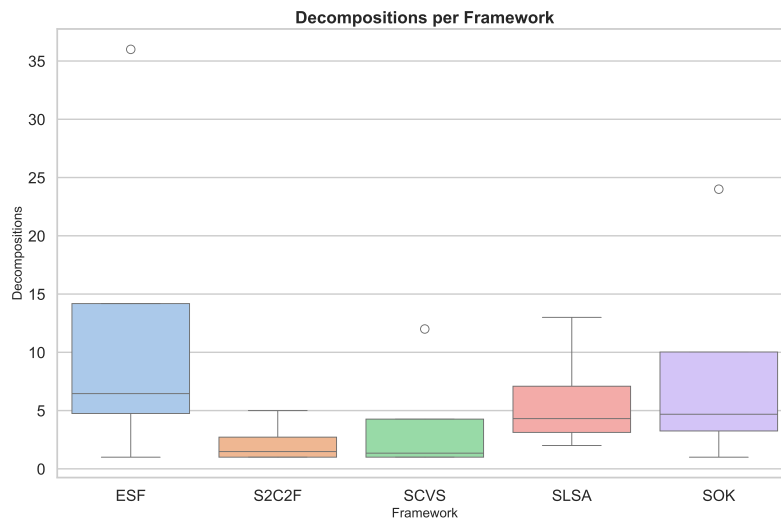

Figure 4.1: Box plot of the number of decompositions per guideline across frameworks.

Table 4.1 and Figure 4.1 clearly highlight that the ESF framework accounts for the majority of decomposed guidelines. This is mainly due to the relatively high number of ESF guidelines combined with their tendency to include multiple sub-guidelines. In contrast, the guidelines extracted from S2C2F and SCVS are predominantly atomic and, in many cases, do not require further decomposition.

4.1.1.2 Text Character Counts of Guidelines

Figure 4.2 shows a box plot of the non-decomposed guideline text character counts across frameworks. The figure highlights that guideline lengths vary widely, with some frameworks averaging over 1000 characters per guideline while others remain much shorter.

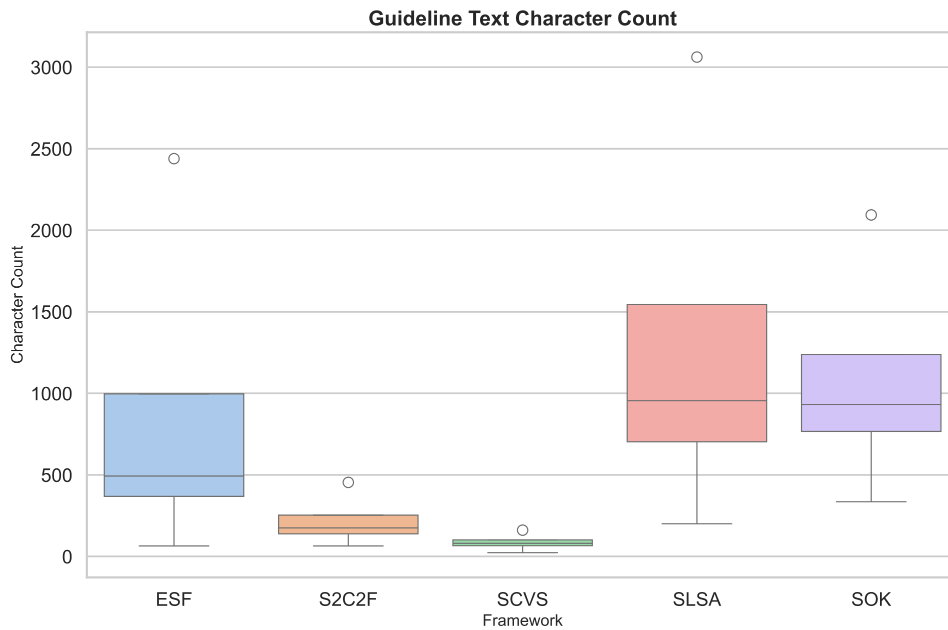

Figure 4.2: Box plot of guideline text character counts across frameworks.

Figure 4.3 displays the distribution of character counts for the decomposed texts. The average character count for decompositions falls between roughly 64 and 99 characters, showing a relatively consistent granularity in the decomposition process across frameworks.

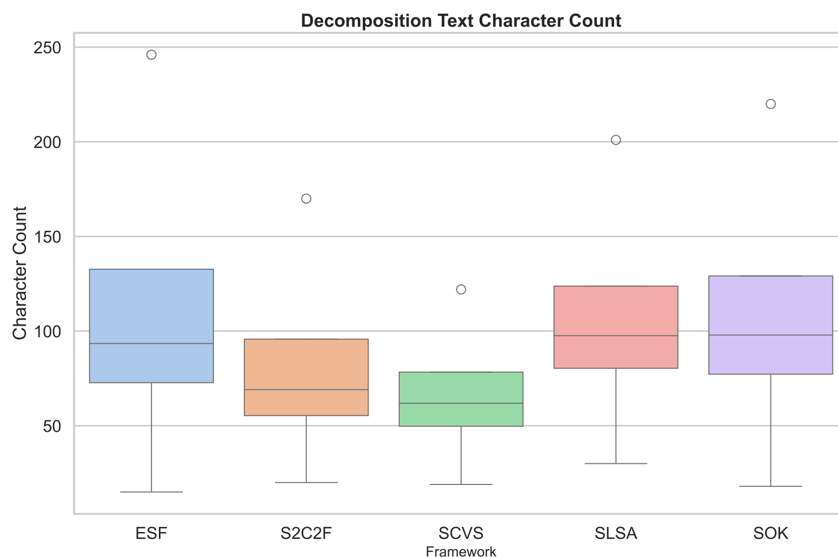

Figure 4.3: Box plot of decomposed text character counts across frameworks.

The character count of the guidelines in Figure 4.2 aligns with the number of decompositions observed in Figure 4.1. Frameworks with a relatively higher number of decompositions per guideline, such as ESF, SLSA, and SOK, also exhibit longer text lengths. This suggests that longer guidelines generally tend to contain multiple

sub-guidelines, rather than simply being more detailed or specific. However, Figure 4.3 shows that the decompositions from S2C2F and SCVS tend to be shorter, which indicate that they are less detailed or specific.

4.1.2 Semantic Overlap of Decompositions

In this section, the results of how the theoretical frameworks' decomposed guidelines overlap in terms of their semantic similarity are presented. The results generated by comparing the cosine similarity of the decomposed guidelines. Figure 4.4 illustrates how many of the decomposed guidelines from the source framework have a cosine similarity score of at least *threshold* (ranging from 0.5 to 0.8) with at least one decomposed guideline in the target framework. The percentages therefore represent how much of the source framework's guidelines is covered by the target framework for that threshold. A higher threshold generally means that the matches are more closely semantically similar. Table 4.2 presents the top 10 pairs of decomposed guidelines with the highest semantic similarity across the different frameworks.

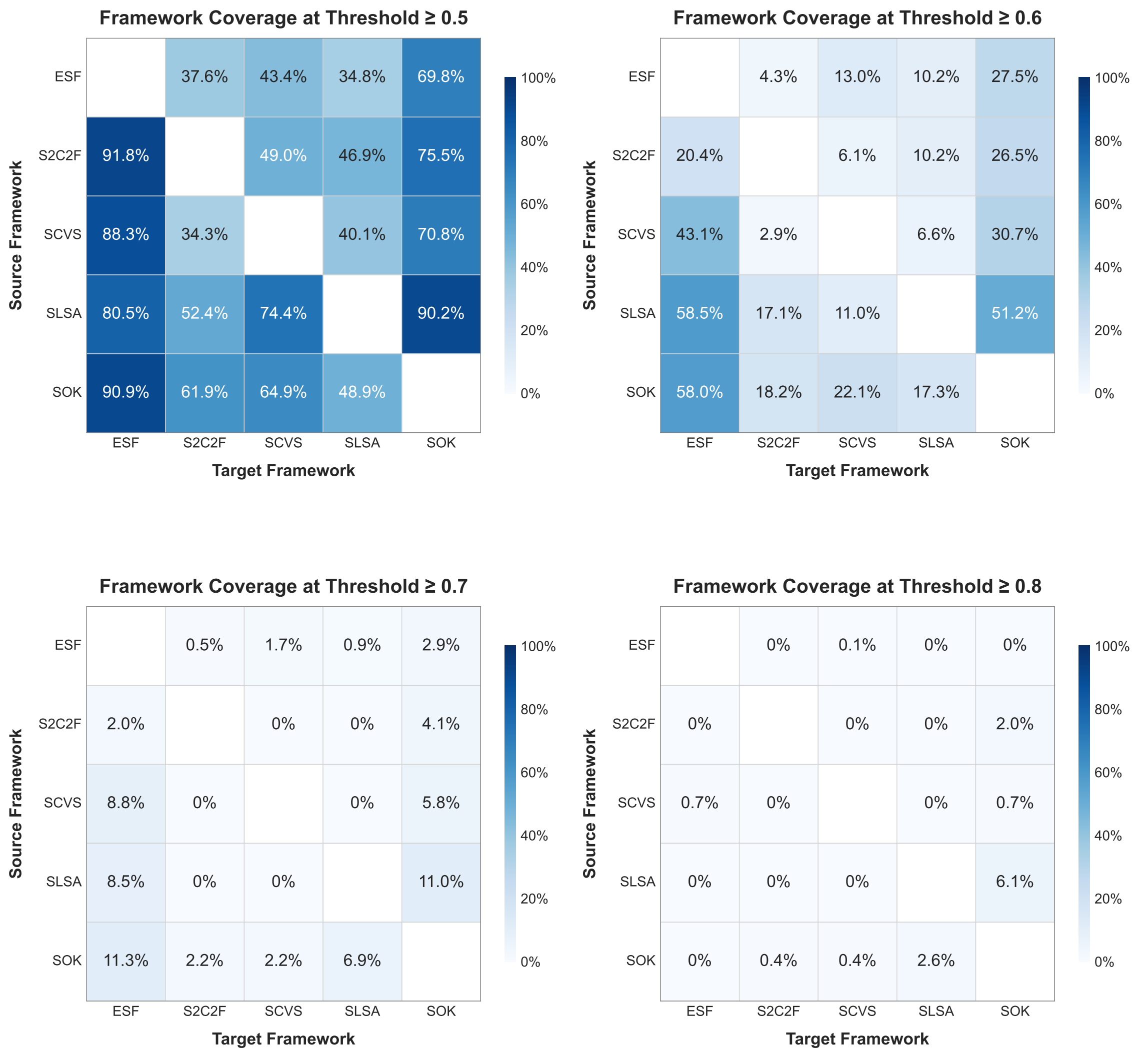

Figure 4.4: Coverage of source framework guidelines by target frameworks at different thresholds for cosine similarity.

Table 4.2: The top 10 most semantically similar statements (cosine similarity).

Source Text	**Target Text**	**Source**	**Target**	**Score**
SBOM is timestamped.	SBOM can be timestamped.	SCVS	SOK	0.95
The build service should ensure that individual builds run in an isolated environment.	The build platform MUST ensure that each build runs in an isolated environment.	SOK	SLSA	0.87
Completeness of resolved dependencies in the provenance is best effort.	Individual dependencies can be described in regard to provenance.	SLSA	SOK	0.82
One build should not be able to influence other builds.	The build platform MUST prevent overlapping builds from influencing each other.	SOK	SLSA	0.81
Consumers MUST be able to validate authenticity.	Consumers can check the signatures to verify provenance.	SLSA	SOK	0.81
SBOM should contain all primary (top level) components of the final product.	SBOM contains a complete inventory of all components the SBOM describes.	ESF	SCVS	0.80
Software composition analysis (SCA) tools should be applied to the software deliverable from the third-party.	Downstream Users can use Software Composition Analysis tools to inspect the components of a software product.	ESF	SOK	0.79
SBOM has been signed by supplier.	SBOM can be signed.	SCVS	SOK	0.77
Best practices should be implemented to protect any secrets associated with the build pipeline.	Build platform implementations MUST implement proper management of cryptographic secrets.	ESF	SLSA	0.76
Release criteria should include correlating a Software Bill of Materials (SBOM).	Software bill of materials are generated for publicly available applications.	ESF	SCVS	0.75

Table 4.2 and Figure 4.5 indicate that the semantic analysis model identifies very few closely similar decompositions across frameworks. Moreover, Table 4.2 shows that the decompositions with high similarity scores tend to be similar in length, suggesting that text length is a significant factor influencing the models assessment of semantic similarity.

4.1.3 Labeling Decompositions

To facilitate a structured analysis of the decomposed guidelines, each decomposition was assigned one or more labels that categorize its content within a specific security or software supply chain domain. These labels provide a way to compare and analyze guidelines across different frameworks based on their thematic focus.

4.1.3.1 Labels

Table 4.3 presents the set of labels that could be assigned to each decomposed guideline. Each label represents a distinct aspect of security or supply chain practices, ranging from authentication mechanisms to policy governance.

Table 4.3: The labels and their corresponding description.

Label	Description
Architecture, Design & Documentation	The structure, components, and interactions of software systems, including design principles, patterns, and architectural decisions. Also includes system documentation, technical guides, and descriptions of how an application is built and maintained.
Authentication, Cryptography & Secrets Management	Authentication mechanisms, encryption, key management, and protection of sensitive information, including multi-factor authentication (MFA) and credential storage.
Build Process	Infrastructure, pipelines, and workflows for building software, including security controls for protecting the build environment and ensuring artifact integrity.
Dependency & Package Consumption	Selection, management, and integration of third-party libraries, frameworks, and dependencies, ensuring secure sourcing and maintenance.
Development Process Controls	Security and quality assurance measures in development, including branch protections, code reviews, access controls, and contribution workflows.
Incident Response & Support Lifecycle	Management of security incidents, breaches, and support processes, including response plans, remediation, and end-of-life procedures.
Known Vulnerability Handling & Patching	Management and mitigation of known security vulnerabilities, including patching, security updates, and remediation strategies.
Policy, Governance & Training	Establishment and enforcement of security policies, governance frameworks, and training programs for improving security awareness and compliance.
Provenance, Attestations & Reproducible Builds	Tracking the origin and authenticity of software artifacts, ensuring build integrity, and supporting reproducible builds.
QA, Testing & Security Scanning	Methods for evaluating software quality and security, including automated and manual testing, security scanning, and code integrity validation.
Release & Distribution	Secure publishing, deployment, and distribution of software, ensuring artifact integrity and protecting users from supply chain risks.

Label	Description
Secure Development Environment	Security measures for development workstations, tools, and environments to protect code, prevent unauthorized access, and reduce risks.
Software Bill of Materials (SBOM)	Creation, management, and use of SBOMs to provide a detailed inventory of software components, dependencies, and their origins.
Software Composition Analysis	Evaluation of software components, libraries, and dependencies to understand their structure, relationships, and security impact.
Threat Modeling & Risk Assessment	Identification, evaluation, and mitigation of security threats and risks. Includes assessing attack vectors and designing countermeasures.

4.1.3.2 Label Coverage Across Frameworks

Figure 4.5 presents the distribution of labels, as defined in Table 4.3, across the decomposed guidelines of each framework. Each cell represent the proportion of decompositions within a given framework that were assigned the corresponding label. The figure highlights the focus areas of each framework: while frameworks such as ESF display a relatively even distribution of content across categories, others – such as S2C2F and SLSA – exhibit a more concentrated emphasis on specific thematic areas.

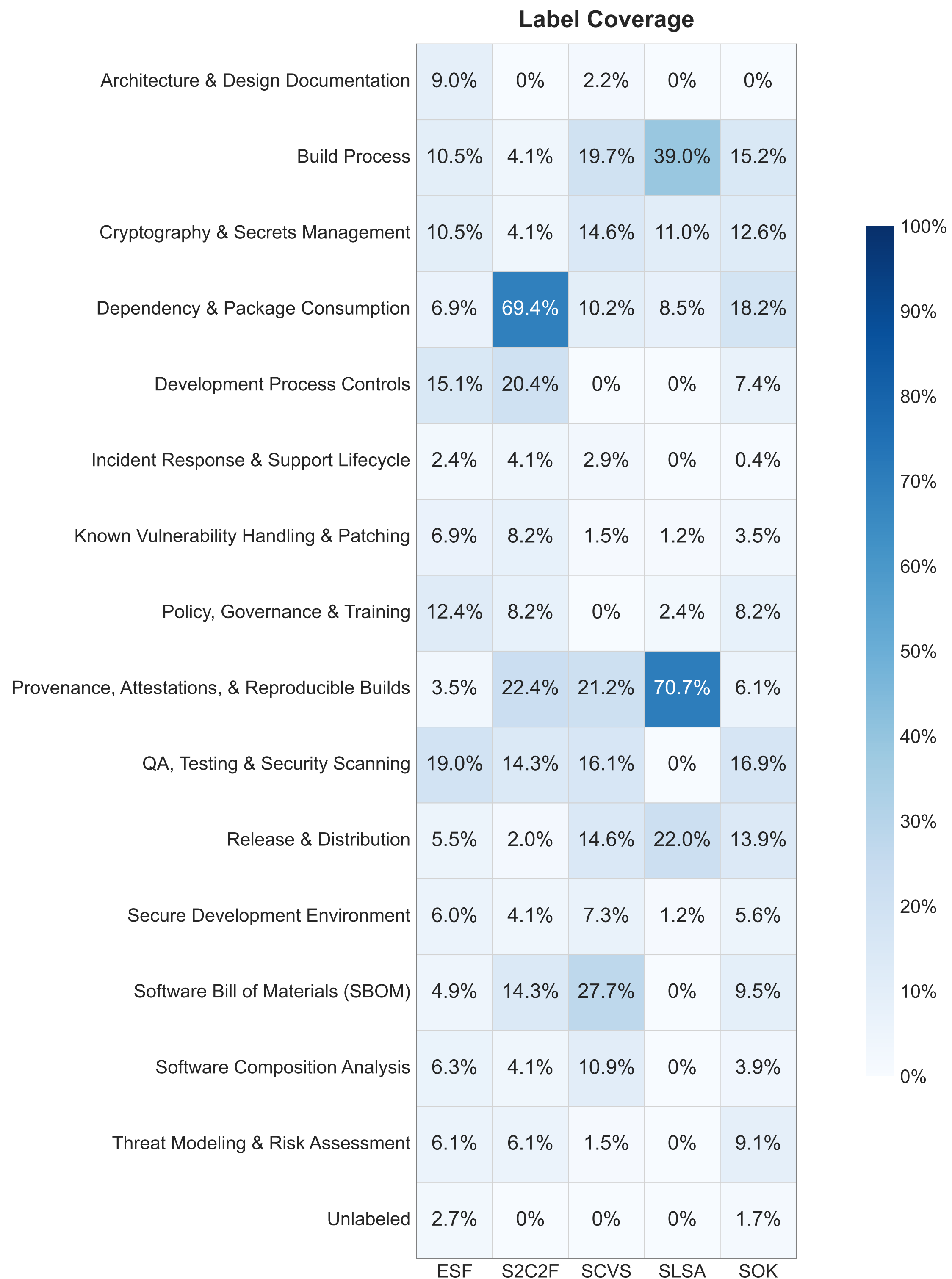

Figure 4.5: Heatmap of the label distribution across frameworks.

4.1.3.3 Distribution of Number of Labels per Decomposition

Table 4.4 presents the distribution of labels assigned to the decompositions within each framework. The results suggest that the thematic composition of the decom-

posed guidelines varies across frameworks. Most of the ESF decompositions are associated with only a single label, whereas more than 60% of the S2C2F decompositions are linked to multiple labels. This indicates a broader thematic scope in the S2C2F decompositions compared to those from ESF. A similar pattern can be observed in the SLSA framework, where the majority of decompositions are associated with two labels.

Table 4.4: The label distribution across frameworks

Number of Labels	**ESF**		**S2C2F**		**SCVS**		**SLSA**		**SOK**	
	#	%	#	%	#	%	#	%	#	%
0	22	2.7%	0	0.0%	0	0.0%	0	0.0%	4	1.7%
1	591	71.9%	19	38.8%	80	58.4%	38	46.3%	156	67.5%
2	191	23.2%	18	36.7%	45	32.8%	42	51.2%	68	29.4%
3	18	2.2%	12	24.5%	12	8.8%	2	2.4%	3	1.3%
Decompositions	**822**		**49**		**137**		**82**		**231**	
Distributed Labels	**1027**		**91**		**206**		**128**		**301**	

4.1.4 Framework Coverage and Suitability Analysis

The comparative analysis, including guideline decomposition (Section 4.1.1) and thematic labeling (Section 4.1.3, particularly Figure 4.5), allows a better understanding of each selected software supply chain security framework's primary focus, key coverage areas, and ideal use cases. Table 4.5 below summarizes these findings, offering insights into which framework is best suited for specific organizational needs and security objectives. The "Key Coverage Areas" are derived from the label distribution analysis (Figure 4.5), highlighting the thematic categories where each framework dedicates a significant portion of its guidelines.

Table 4.5: SSCS framework coverage, focus, and suitability

Framework	Primary Focus	Key Coverage Areas	Strengths / Distinguishing Features	Best Suited For
ESF	Comprehensive developer-centric security practices across the SDLC.	Broad coverage: QA, Testing & Security Scanning; Development Process Controls; Policy, Governance & Training; Build Process; Cryptography & Secrets Management.	Most comprehensive in terms of guideline count and breadth of topics. Government-backed. Detailed, though sometimes lengthy, recommendations.	Organizations seeking extensive, developer-oriented guidelines for overall SDLC security improvement; establishing a foundational security posture.
S2C2F	Secure consumption of Open-Source Software (OSS) and third-party components.	Dependency & Package Consumption (dominant); Provenance, Attestations & Reproducible Builds; Development Process Controls.	Highly specialized on the intake and management of external software. Tiered maturity model for incremental adoption.	Organizations heavily reliant on OSS and needing to establish robust processes for ingesting, vetting, and managing third-party dependencies.
SCVS	Verification of software components and their supply chain security.	Software Bill of Materials (SBOM); Provenance, Attestations & Reproducible Builds; Build Process; QA, Testing & Security Scanning.	Component-level verification focus. Tiered maturity levels. OWASP community-driven. Practical and actionable controls.	Organizations needing to verify the security of individual software components, implement SBOM practices, and incrementally improve supply chain security.

Continued on next page

Table 4.5: SSCS framework coverage, focus, and suitability (continued)

Framework	Primary Focus	Key Coverage Areas	Strengths / Distinguishing Features	Best Suited For
SLSA	Integrity of software artifacts and the build process; provenance.	Provenance, Attestations & Reproducible Builds (dominant); Build Process; Release & Distribution.	Strong emphasis on verifiable build provenance and artifact integrity. Tiered levels (0-3) to define increasing security assurances.	Organizations focused on securing their build pipelines, preventing tampering of software artifacts, and verifying software origins.
SOK	Academic taxonomy of attacks on OSS supply chains and corresponding safeguards.	Broad coverage mapping to safeguards: Dependency & Package Consumption; QA, Testing & Security Scanning; Build Process.	Attack-centric perspective. Provides a structured understanding of threats and links them to mitigations. Academic rigor.	Researchers, threat modelers, and organizations wanting to understand the attack landscape in-depth and map existing controls to known attack vectors.

Interpretation of Findings:

The analysis presented in 4.5, informed by the preceding quantitative results, highlights that no single framework is universally "best". Instead, their suitability depends on the specific context, priorities, and existing maturity of an organization.

- **ESF** offers the broadest, most developer-focused guidance, making it a strong starting point for comprehensive SDLC security.
- **S2C2F** excels in the niche of secure open-source software consumption, a significant area for many modern development practices.
- **SCVS** provides a structured, component-centric approach with clear maturity levels, ideal for targeted verification efforts.
- **SLSA** is unparalleled in its focus on build integrity and provenance, crucial for ensuring the authenticity of software artifacts.
- The **SOK** paper delivers relevant academic insight, useful for grasping the "why" behind many safeguards by detailing the attack vectors they mitigate.

Organizations may find value in consulting multiple frameworks. For instance, a team might use SLSA to secure its build pipeline, S2C2F to manage dependencies, and refer to ESF for broader secure coding and organizational practices, all while using the SOK to understand the threat landscape. The decomposition and labeling performed in this thesis facilitate such cross-referencing by breaking down guidelines into more comparable units.

4.2 Feasibility of the End-to-End Attestation Process

This section presents outcomes from the feasibility study (detailed in Section 3.2.7), which aimed to demonstrate the practical viability of an end-to-end automated compliance checking and attestation process. The study involved applying the PoC system to five open-source Node.js projects against selected guidelines from SCVS, S2C2F, and SOK (Table 3.1 and 3.2).

4.2.1 Generation and Structuring of Compliance Attestations by the PoC System

The initial phase of the feasibility study focused on the compliance data produced and organized by the automated PoC system. The system's operation against the target projects resulted in SBOMs containing structured and signed compliance attestations, demonstrating several key capabilities:

- **Automated Check Execution and Varied Guideline Coverage:** The system executed its automated checkers against the target projects. These checks, covering diverse guidelines (as detailed in Table 3.1 and 3.2), generated specific compliance data. Table 4.6 summarizes the resulting conformance and confidence scores. For instance, the system's SCVS 2.3 (SBOM ID) checks consistently indicated full conformance, while its S2C2F ENF-1 (Secure Package Source) checks correctly identified non-conformance for most projects due to unpinned dependency versions. The "Observations on Selected Guideline Checks" (Section 4.2.1.1) provides further detail on these outputs, illustrating the system's processing of varied requirements.

- **Structured Attestation Data in CycloneDX Format:** The system automatically transformed the raw outputs from its checkers into CycloneDX attestation constructs (`Evidence`, `Claims`, `Targets`). These constructs were then embedded by the system in each project's SBOM, adhering to the structure outlined in Section 3.2.1.

- **Integrated Cryptographic Signing:** As a final step in its generation process, the system cryptographically signed the generated `Evidence`, `Claim`, `Target`, and overall `Attestation` objects within the SBOMs using JWS, per the approach defined in Section 3.2.4.

The outcome of this automated process was the production of signed SBOMs, where each SBOM contained structured compliance attestations derived from the system's internal checks.

Table 4.6: Summary of automated compliance check outcomes for the selected projects. (first number = Conformance Score, Cnf. = Confidence Score)

Project	SCVS 2.3 (SBOM ID)	SCVS 3.3 (CI Pipeline)	S2C2F ENF-1 (Secure Source)	SOK SG-010 (Script Disable)	SOK SG-016 (Branch Protect)	SOK SG-036 (Vuln Scan)
Express.js	1 (Cnf: 1.0)	0 (Cnf: 0.6)	0 (Cnf: 0.83)	0 (Cnf: 0.8)	1 (Cnf: 1.0)	1 (Cnf: 0.9)
Axios	1 (Cnf: 1.0)	0 (Cnf: 0.6)	0 (Cnf: 0.77)	0 (Cnf: 0.8)	1 (Cnf: 1.0)	1 (Cnf: 0.9)
Fastify	1 (Cnf: 1.0)	0 (Cnf: 0.6)	0 (Cnf: 0.83)	1 (Cnf: 1.0)	1 (Cnf: 1.0)	1 (Cnf: 0.9)
NestJS	1 (Cnf: 1.0)	0 (Cnf: 0.6)	1 (Cnf: 0.93)	0 (Cnf: 0.8)	1 (Cnf: 1.0)	1 (Cnf: 0.9)
React	1 (Cnf: 1.0)	0 (Cnf: 0.6)	0 (Cnf: 0.77)	0 (Cnf: 0.8)	1 (Cnf: 1.0)	0 (Cnf: 0.7)

4.2.1.1 Observations on Selected Guideline Checks

- **SCVS 2.3 (SBOM ID):** The tool reported a conformance score of 1 (conformant) with a confidence of 1.0 for all projects. The SBOMs processed by the tool include a `serialNumber` field, fulfilling the checker's criteria.
- **SCVS 3.3 (CI Pipeline):** For all five projects, the tool reported a conformance score of 0 (non-conformant) with a confidence of 0.6. The rationale provided in the SBOM attestations (e.g., for Express.js: "Build job not found by keyword/command matching...") indicated that the PoC checker's heuristics did not identify a definitive "build job." Manual inspection of the projects' CI/CD configurations confirmed the presence of CI pipelines in all cases.
- **S2C2F ENF-1 (Secure Package Source):**
  - For NestJS, the tool reported conformance (1) with high confidence (0.93).
  - For Express.js (0, Cnf: 0.83), Fastify (0, Cnf: 0.83), Axios (0, Cnf: 0.77), and React (0, Cnf: 0.77), the tool reported non-conformance. The `Evidence` objects in the SBOMs primarily cited unpinned dependency versions as the reason, which was consistent with manual verification of their `package.json` files. The tool also reported that the Express.js project is missing a lock file.
- **SOK SG-016 (Branch Protection):** The tool reported conformance (1) with high confidence (1.0) for all five projects. The `Evidence` in the SBOM attestations indicated that GitHub API checks confirmed protection was enabled on their main/master or other common sensitive branches, aligning with manual repository setting checks.
- **SOK SG-036 (Vulnerability Scanning in CI):**
  - For Express.js, Axios, Fastify, and NestJS, the tool reported conformance (1) with high confidence (0.9), identifying known vulnerability scanners (e.g., "CodeQL," "dependabot") in their CI workflows. This was consistent with manual inspection of CI files.
  - For React, the tool reported non-conformance (0) with a confidence of

0.7. The rationale indicated "GitHub Actions workflows exist but no security-related keywords found."

These observations highlight that the PoC tool generated specific, evidence-backed attestations based on its predefined checking logic. The generated SBOMs contain the detailed rationale and evidence for each of these automated assessments.

4.2.2 Verification and Presentation of Embedded Attestations

The SBOMs containing embedded compliance data were processed using the visualization and verification tool, designed as described in Section 3.2.6. This step assessed the ability to consume and cryptographically verify the generated attestations.

All SBOMs produced in Section 4.2.1 were successfully parsed by the tool. The tool rendered the compliance information hierarchically, allowing navigation from overall frameworks to specific claims and evidence. This demonstrated that the structured attestation data is human-interpretable when presented appropriately.

The cryptographic signature verification capability of the tool was tested. When supplied with the correct public keys, the JWS signatures on attestations, claims, evidence, and targets were consistently verified, confirming data integrity and authenticity (as exemplified by the tool's output features shown previously in methods, such as in Figure 3.6). Conversely, the tool correctly indicated failures or discrepancies when faced with scenarios such as incorrect keys or missing signatures (aligning with behaviors illustrated in methods, for instance, in Figure 3.7 and Figure 3.8).

These outcomes indicate that the attestations generated by the PoC system are both consumable and their cryptographic integrity verifiable, which are crucial for establishing trust.

4.2.3 Assessment of Overall Feasibility

The outcomes from the feasibility study indicate the operational viability of an end-to-end compliance attestation approach. The key findings from the application of the PoC system are:

- Automated generation of compliance data for varied guideline types was demonstrated, with this data structured within SBOMs using the CycloneDX attestation model.
- Cryptographic signatures were integrated into the automated data generation process.
- The SBOMs containing embedded attestations were shown to be machine-parsable, enabling their presentation in a human-understandable format and the verification of embedded attestations, including their cryptographic signatures, by a dedicated tool.

These results support the practical application of the described mechanism for generating, embedding, and verifying compliance attestations, providing a basis for further investigation in this domain.

5

Discussion

5.1 Discussion of Comparative Analysis

The first major contribution of this thesis involved a systematic comparative analysis of five prominent software supply chain security frameworks: ESF, S2C2F, SCVS, SLSA, and the SOK taxonomy. This section presents a detailed discussion of the findings and methodological considerations arising from this analysis. We begin by examining the insights gained and challenges encountered during the initial guideline decomposition process. Subsequently, we evaluate how similar the guidance is between frameworks using the results of the semantic comparison, as well as reflecting on the performance of the similarity model used. This is followed by an in-depth discussion of our thematic analysis, covering the labeling process, the resultant distribution of themes, and the identification of consensus areas and potential gaps in current SSCS guidance. Finally, this section combines these findings to propose pathways towards a potential unification of SSCS frameworks.

5.1.1 Insights and Challenges in Guideline Decomposition

The initial step of our comparative analysis involved understanding the inherent structure of software supply chain security guidelines and then standardizing them through decomposition. This process revealed both key characteristics of the existing frameworks and inherent challenges in transforming them for systematic comparison.

5.1.1.1 The Variable Nature of Source Guidelines: Length and Granularity

A primary observation was the notable variation in the length and granularity of original guideline texts across the analyzed frameworks. As illustrated in Figure 4.2, frameworks such as SLSA and SOK present significantly longer original guidelines, contrasting with the more concise nature of SCVS and S2C2F. This variance suggests fundamental differences in writing style, intended level of detail, and how requirements are formulated, making direct one-to-one comparisons of the original texts inherently difficult and necessitating a normalization step.

Decomposition aimed to address this by breaking guidelines into more atomic statements. While this led to a more uniform character count for individual decomposed statements (Figure 4.3), differences in median length persisted. Frameworks like SLSA, SOK, and ESF still yielded slightly longer decomposed statements, potentially reflecting a higher level of detail or technical specificity in their source material. Conversely, SCVS and S2C2F produced shorter decomposed statements, likely due to simpler or more general phrasing in their original guidelines. This difference in decomposed length, even after normalization, is an important insight: it underscores that not all frameworks operate at, or can be perfectly reduced to, the same level of granularity, which has implications for both comparison and potential unification.

5.1.1.2 Challenges in the Decomposition Process

While decomposing original guidelines into granular, atomic statements enabled a detailed comparison, this methodological step also had some specific limitations. These challenges primarily concern the consequences of manual processing and the interpretation of logical structures within the source guidelines.

5.1.1.2.1 The Human Factor in Manual Decomposition The manual breakdown of compound statements, especially from larger and more complex texts, inherently carries the risk of introducing inconsistencies or unintentional deviations from the original guideline's intent. Misinterpretation or subtle shifts in meaning can occur, particularly if the source text lacks clarity or conciseness. This subjectivity, a recognized aspect of qualitative data processing, could potentially influence the outcomes of subsequent analyses, such as semantic comparisons. While efforts like those by Ding et al. [20] demonstrate the potential of large language models for policy extraction, their applicability to the nuanced decomposition of security guidelines into independent, actionable units remains an area for future investigation. Furthermore, building on the work by Wanner et al [21], it may be possible to leverage LLMs to accurately decompose the extracted guidelines into their atomic counterparts. Automating this process could significantly enhance reproducibility and mitigate human bias.

5.1.1.2.2 Handling Logical Structures: The AND/OR Dilemma One limitation encountered in the decomposition process was the handling of statements that included *"or"* conditions, for example, *"SBOM has been signed by publisher, supplier, or certifying authority"*. In favor of decomposing the statements into smaller statements these formulations were transformed into separate guidelines, *"SBOM has been signed by publisher"*, *"SBOM has been signed by supplier"*, and *"SBOM has been signed by certifying authority"*. This choice results in an implicit *"AND"* relationship, where both options appear as separate mandatory guidelines rather than as alternatives. Consequently, the flexibility originally intended by the *"or"* condition was lost in the decomposition, as well as some of the original intent. However, this approach was still adopted to be able to better highlight the overlapping content between frameworks in more granularity during the semantic comparison. While this approach may have provided a more detailed comparison in terms of thematic content, it implies that these specific decompositions might be less applicable for constructing a new, unified framework that perfectly preserves the original disjunctive logic of the source material.

5.1.2 Evaluating Semantic Relationships Between Frameworks

Once guidelines were decomposed, the next step involved assessing the semantic relationships between them across different frameworks. This was achieved by measuring semantic similarity, which provided insights into inter-framework overlap and also highlighted the capabilities and nuances of the similarity model itself.

5.1.2.1 Observed Inter-Framework Overlap

The analysis of decomposed guidelines, as depicted in Figure 4.4, reveals a clear, though threshold-dependent, overlap between the different frameworks. At a broad semantic threshold (e.g., cosine similarity ≥ 0.5), a high degree of overlap is evident. For instance, ESF encapsulates more than 80% of the content of all other frameworks at this level, suggesting that, in general terms, the frameworks address many common topics or conceptual areas.

However, this apparent consensus diminishes rapidly as the similarity threshold increases. The steep decline in overlap percentage with higher thresholds indicates that while frameworks may cover similar thematic ground, the specific language, phrasing, and granularity of individual guidelines are often not closely aligned.

5.1.2.2 Performance and Limitations of the Semantic Similarity Model

The MPNet-based cosine similarity model was employed to quantify the degree of semantic similarity, where a score of 1 represents near-identical meaning and 0 indicates no meaningful relationship. The model's effectiveness and its interpretative nuances can be illustrated by examining its performance on specific examples. Consider the SoK guideline:

> *"The creation of SBOMs can be automated"*

When compared against guidelines from other frameworks, this yielded the following highest similarity scores:

- S2C2F: *"Generate SBOMs for your software"* (0.82)
- SCVS: *"SBOM creation is automated"* (0.73)
- ESF: *"An SBOM should contain all primary (top-level) components of the final product."* (0.59)
- SLSA: *"The build platform should provide provenance generation."* (0.39)

These scores suggest that the model performs reasonably well in identifying semantically related guidelines. A score ≥ 0.7 generally signifies a close, though not necessarily exact, match, capturing significant conceptual overlap despite minor differences in phrasing. For instance, both the S2C2F and SCVS guidelines are clearly related to the SoK example.

However, this example also illuminates important characteristics and limitations of the model. While the S2C2F guideline achieved the highest numerical score (0.82), the SCVS guideline (0.73) could be argued as a stronger conceptual match due to the shared term *"automated,"* which directly aligns with the SoK guideline's core assertion. This discrepancy highlights a key takeaway: while cosine similarity provides a valuable numerical measure of closeness, it does not always perfectly determine which match is most semantically relevant or contextually appropriate. The model can be influenced by shared vocabulary and sentence structure beyond the core meaning.

The interpretation of lower scores provides further insight. The ESF match (0.59), while sharing the general topic of SBOM, does not address the automation aspect, suggesting that scores around 0.6 may indicate thematic connection but not necessarily functional or operational equivalence. The SLSA match (0.39) demonstrates almost no meaningful similarity to the SOK guideline's specific point, indicating that scores around 0.4 or lower likely represent a lack of direct conceptual alignment.

These findings suggest that while cosine similarity offers a useful approximation of how closely the content of different frameworks aligns, its scores should be interpreted with caution. Notably, scores higher than 0.8 may still fall short of exact matches, and lower-scoring guidelines might sometimes better capture the intended conceptual meaning.

5.1.3 Thematic Analysis: Labeling, Distribution, Consensus, and Gaps

This section first presents the key thematic insights derived from the labeling analysis, encompassing the distribution of themes within individual frameworks, identified areas of consensus, and potential gaps in the current landscape of SSCS guidance. Subsequently, it discusses the inherent limitations of the labeling approach employed.

5.1.3.1 Label Distribution

As illustrated in Figure 4.5, ESF demonstrates a broad, even distribution of guidelines across multiple security topics. This comprehensive approach indicates that the framework is designed to cover a wide spectrum of issues. Such diversity in coverage supports the view that ESF aims to address multiple facets of secure software development, although the depth in each area may vary.

In contrast, S2C2F is highly specialized, with the vast majority of its guidelines focused on *Dependency & Package Consumption*. This emphasis is consistent with the frameworks core objective of securing the software supply chain through strict controls on third-party components. While S2C2F includes some guidelines related to build provenance and development process controls, these are secondary to its principal concern of managing external dependencies securely.

SCVS is distinguished by its focus on mitigating risks associated with third-party components. A significant portion of its guidelines is devoted to ensuring transparency and accountability, particularly through the use of Software Bill of Materials and secure build practices. Interestingly, SCVS places minimal emphasis on *Development Process Controls* and *Policy, Governance & Training*, suggesting that it relies on the assumption that developers maintain robust internal controls. In effect, SCVS entrusts much of the security posture to the developers environment and practices, rather than prescribing detailed organizational policies.

SLSA is narrowly tailored to the security of the build process. It allocates the majority of its guidelines to *Provenance, Attestations & Reproducible Builds* and the *Build Process* itself. This focus is in line with its objective of ensuring that

every software artifact is produced through a trusted and verifiable pipeline. The inclusion of some guidelines on *Release & Distribution* further underscores that SLSA views the build process as integral not only to production but also to the subsequent deployment of secure software. However, its narrow scope means that aspects such as organizational governance and incident response receive little to no attention.

The SOK framework adopts a more balanced approach but is predominantly left-focused. It provides moderate coverage for early-stage activities like dependency management, QA/testing, and the build process. However, SOK is less comprehensive regarding later-stage processes such as *Incident Response*, or *Known Vulnerability Handling & Patching*. This focus suggests that SOK prioritizes proactive security measures, aiming to mitigate risks before software deployment, over the development of extensive post-deployment or organizational process controls.

5.1.3.2 Consensus and Gaps of Guidance

The analysis of guideline coverage across the frameworks reveals a broad consensus on certain core areas relevant to software supply chain security. Categories such as *Build Process*, *Dependency & Package Consumption*, *QA, Testing & Security Scanning*, and *Development Process Controls* consistently account for a significant share of guidelines. This suggests that these areas are widely recognized as critical for securing the software supply chain.

In contrast, some categories exhibit lower coverage. For example, *Architecture, Design and Documentation* is covered in only one framework at a level of 9.0% (ESF) and minimally in SCVS (2.2%), with the remaining frameworks not dedicating any guidelines to this category. This low coverage may reflect a deliberate choice, as such areas are sometimes considered out-of-scope for software supply chain security, focusing instead on broader software development practices.

Similarly, *Incident Response & Support Lifecycle* and *Known Vulnerability Handling & Patching* receive modest coverage, with guideline percentages ranging from approximately 0.4% to 4.1% and 1.2% to 8.2%, respectively. This pattern indicates that the current frameworks emphasize proactive, pre-release measures over reactive strategies. Although these areas are essential for managing supply chain risks after deployment, their limited coverage might suggest an intentional prioritization or an area where future frameworks could expand their scope.

The *Secure Development Environment* category also shows an uneven distribution. While ESF allocates 6.0% of its guidelines to this domain, other frameworks such as SLSA offer as little as 1.2% coverage. Although the absolute number of guidelines may seem substantial, the percentage share implies that the broad spectrum of threats in this area might not be comprehensively addressed across all frameworks.

In summary, while there is strong consensus on key areas central to software supply chain security, the relative coverage indicates that some categories—especially those potentially seen as peripheral (e.g., *Architecture, Design and Documentation*)—receive less attention. This disparity in guideline distribution highlights both the common focus areas and the potential gaps or deliberate exclusions based on

differing interpretations of the scope of software supply chain security.

5.1.3.3 Limitations of the Labeling Approach

While the this labeling approach was very useful for understanding the scope and focus of each framework, the chosen inductive approach has some inherent limitations that warrant discussion. These primarily revolve around the human element in the labeling process and the potential scope limitations of the derived label set.

Human Factor and Subjectivity in Label Assignment: The manual process of assigning thematic labels to decomposed guidelines (Section 3.1.5) is inherently subjective, presenting a limitation that could influence the study's findings. The core of this process involves researchers making judgments about which label(s) best represent the semantic content of each decomposed guideline. Deciding whether a specific guideline aligns sufficiently with a label's definition (Table 4.3) is a subjective act. Even with an iterative refinement process aimed at creating clear definitions and achieving consensus, different researchers, or the same researcher at different times, might interpret the nuances of a guideline or a label definition slightly differently.

This subjectivity in assigning labels can directly impact the quantitative representation of thematic coverage. Aggregated across many such labeling assignments, this could slightly skew the perceived prominence of certain themes within a framework, as reflected in the label distribution heatmap (Figure 4.5) and the distribution of labels per decomposition (Table 4.4). Consequently, the analysis of thematic focus, consensus, and gaps might be influenced by these cumulative subjective categorizations. Furthermore, the decision of whether a single guideline warrants one or multiple labels is also subjective. A tendency to assign more labels might make a framework appear more multifaceted, while a more conservative approach might make it seem more focused. This subjective element in determining labeling granularity can, therefore, affect the interpretation of a framework's breadth versus depth.

While the systematic approach, including iterative refinement by Mayring [51], was designed to enhance consistency and reduce arbitrary decisions, the fundamental act of mapping complex textual information to predefined categories involves a level of human interpretation. This subjectivity is a recognized characteristic of qualitative content analysis, meaning that the precise quantitative distributions presented should be understood as strong indicators rather than absolute, objective measures.

Label Set Scope and Derivation: The inductive process of deriving labels primarily from the content of the selected frameworks presents a limitation regarding the comprehensiveness of the label set for the *entire* software supply chain security domain. Because the labels emerged from the analyzed frameworks themselves, there is a risk that the final set of 15 labels (Table 4.3) does not fully encompass every conceivable aspect of SSCS. If a specific SSCS concern or best practice is not addressed by any of the five chosen frameworks, a corresponding label might not have been generated during our inductive process.

Consequently, this approach could make it more challenging to identify absolute

gaps in SSCS guidance, meaning areas of SSCS that are missing entirely from the analyzed frameworks and thus from our derived label set. While the analysis can effectively highlight relative gaps among the selected frameworks, it will be less equipped to identify gaps when compared against the theoretical total domain of SSCS.

However, the inclusion of the SOK taxonomy [11] significantly mitigates this concern. As the SOK paper aims to provide a comprehensive taxonomy of attacks and their corresponding safeguards based on a systematic literature review, its inclusion broadens the foundation from which labels were derived. The assumption is that the safeguards identified by Ladisa et al. cover a vast majority of currently known SSCS attack vectors and mitigation strategies. Therefore, labels influenced by these safeguards are likely to be representative of the current state of recognized SSCS concerns. Despite this mitigation, the label set remains fundamentally tied to the knowledge encapsulated within the selected source materials at the time of analysis, and emerging SSCS threats that are not yet widely incorporated into these frameworks may not be adequately represented.

Despite this mitigation, the label set is fundamentally tied to the knowledge encapsulated within the selected source materials at the time of analysis. Emerging SSCS threats or paradigms not yet widely discussed or incorporated into these frameworks might not be adequately represented by the current label set.

An alternative approach would involve using an existing taxonomy or classification system, such as those present in MITRE ATT&CK [52], to label guidelines. This could potentially better identify absolute gaps in SSCS guidance as well as improve alignment with industry terminology and reduce subjectivity. However, existing taxonomies may not be well-suited to capturing the aspects of software supply chain security since they focus on security as a whole, potentially missing niche SSCS areas.

5.1.3.4 Unlabeled Decompositions

Approximately 2% of the decompositions remain unlabeled, demonstrating that our label set is highly effective in capturing the key aspects of software supply chain security across various existing frameworks. However, some decompositions did not appear in our sampling process and were consequently not considered during label creation. For example, consider the guideline:

> “Properly dispose of physical media by destroying it.”

This guideline focuses on physical security measures rather than the digital or process-oriented controls emphasized in our labels. Although our overall approach worked well, this case suggests that a broader sampling process could potentially capture such outliers, offering opportunities for further refinement of the label set.

5.1.4 Towards Unification: Synthesizing Findings and Proposing Pathways

Based on the collective findings in this thesis, ESF is the most comprehensive framework in terms of the number of decomposed guidelines and distribution of content. However, as seen in Figure 4.4 it does not encompass the content of all other frameworks. This suggests an opportunity to combine frameworks for broader coverage. By merging the guidelines from all frameworks, it can provide a more holistic view and allow developers to focus on one framework.

A recent analysis by the European Union reaches a similar conclusion: a concerted consolidation of software-supply-chain-security frameworks is needed to provide unified guidance that maps requirements across frameworks and clarifies how the respective maturity and assurance levels should be applied [53].

5.1.4.1 Bringing Frameworks To The Same Level of Granularity

Our decomposition approach, which decomposes each guideline in all five frameworks into atomic, independent statements, already places the guidelines on a common level of detail and could serve as the backbone for a more precise requirement-to-requirement mapping across frameworks. However, due to the AND/OR problem in Section 5.1.1.2.2, we argue that we would have to re-evaulate our decompositions and keep OR semantics in the decompositions as this keeping these disjunctions intact preserves the authors intent, maintains the flexibility practitioners expect, and still allows cross-mapping because all frameworks are now expressed at a comparable level of detail.

5.1.4.2 From Static PDFs To An Interactive Portal

While ESF's framework is comprehensive, it may not be presented in the most effective format, especially when extended with other frameworks. The recommendations are provided in PDF format, which may be less optimal for digital screens and consumption of this kind of content [54]. The high number of guidelines and categories increases the cognitive load and makes it harder to navigate the framework and to know which recommendations are most relevant for a given project in this current format. An interactive website could enhance usability by offering dynamic navigation, responsive design, and search and filter functionalities.

5.1.4.3 Introducing Maturity Levels and Attack Mappings

Borrowing the tiered approach of SCVS and S2C2F, the unified framework could offer *maturity levels* that give recommendations based on the user's security requirements. By including this, the framework could suggest guidelines for different use-cases with different security needs, from hobby projects to highly sensitive government projects. Furthermore, by mapping each guideline to the specific attacks it mitigates (via the SOK taxonomy), it would both emphasize its importance and educate developers on the underlying threats.

5.1.4.4 Tooling Support For Adoption

By listing vetted tooling and implementation patterns alongside each guideline, the framework can lower the barrier to adoption and promote the consistent, widespread application of secure practices.

5.2 Discussion of PoC Compliance Tool

Following the comparative analysis of SSCS frameworks, this thesis undertook a practical exploration: the design, implementation, and evaluation of a PoC tool for SBOM-based compliance attestation. This section delves into the insights and challenges encountered during this endeavor. We will discuss the practical feasibility of automating the verification of SSCS guidelines, the complexities of translating these guidelines into executable checks, and the inherent limitations observed in the automated PoC system. Furthermore, this discussion will consider the implications of these findings for future tools, including the necessity of hybrid approaches that combine automated and manual attestations.

5.2.1 Feasibility and Limitations of Automating SSCS Guideline Verification

A key finding from our analysis and the development of the PoC tool is that software supply chain security guidelines vary greatly in how easily they can be automatically checked. Here we describe three categories of guidelines that are, with our solution, difficult to automatically check conformance to:

Guidelines Difficult to Verify with Software: Many of the guidelines involve abstract ideas, quality judgments, or specific processes that depend heavily on human decisions, understanding company rules, and knowing the specific situation. Such guidelines are difficult to translate into clear, computer-driven checks.

- **Example:** The ESF guideline, *“Feature creep should not be allowed to compromise product integrity.”* Automatically checking this guideline is very difficult. Ideas like “feature creep” and “product integrity” depend on personal views and specific product goals, risk acceptance, and design principles. These are typically not written in a manner that modern compliance tools can easily understand and utilize for automated checks.

Guidelines Requiring Specific or Deeply Integrated Automation: While other guidelines may appear straightforward, they must be closely aligned with company-specific tools, proprietary systems, and established ways of working to be effectively automated.

- **Example:** The ESF guideline, *“All known security issues should be tracked as product defects in the organizations defect tracking tool.”* Automatically checking this guideline assumes the tool can connect with the company’s chosen defect tracking system (e.g., Jira, Azure DevOps), understand its data based on potentially unique company methods for classifying defects, and link these findings, perhaps with vulnerability scan results. Such a close connection is typically beyond what a general-purpose tool can do, and would rather have to be customized for the company’s situation.

Guidelines with Potential for Better Verification using Large Language Models (LLMs): A further type of guideline involves checking written documents,

unorganized information, or contextual details where traditional computer checks might only do basic checks (e.g., if a file exists). The development of LLMs and advanced text understanding methods offers a promising, though still new, way for more meaningful, situation-aware checking.

- **Example:** The SCVS guideline, *"Documentation exists on how the application is built and instructions for repeating the build."* While standard automation might check if a `README.md` or `BUILD.md` file exists, LLMs could potentially understand the meaning of its content to figure out if it really describes a build process and if the instructions seem clear and thorough. This goes beyond simple file checks to a more quality-focused assessment, thereby offering a better level of checking for documentation-focused requirements.

5.2.2 SBOM Editing Tool

These types of guidelines, as described above, highlight the basic limitations of relying solely on automatic checks for complete software supply chain security compliance. Since many guidelines fall into types that require extensive human review or very specific setups, relying solely on automatic checkers would mean the compliance picture is not complete. This gap means we need an alternative method for attestations made by individuals.

Consequently, the development of a manual SBOM attestation editing tool becomes an important addition to the automated system. Such a tool would allow human experts (e.g., auditors, senior developers, security architects) to:

- Carefully check and confirm compliance with guidelines that are not suitable for, or not included in, automated checks.
- Manually create and edit attestations (Claims, Evidence, and Targets), following the same CycloneDX format used by the automated compliance tool.
- Explain in detail and link to outside evidence (e.g., policy documents, design diagrams, manual review notes) that supports their attestations, which isn't from software checks.

Combining these manual attestations with the proposed digital signing method enables a solution in which the PoC tool signs its machine-generated attestations using its own private key, while users of the manual attestation tool digitally sign their contributions with separate, personal keys. This two-key system offers:

- **Clear Origin Tracking:** It clearly shows the difference between computer-made and human-made attestations.
- **Improved Truthfulness and Responsibility:** Each attestation is connected to an identity that can be confirmed, whether that is an automated system or a human expert.

5.2.3 Limitations of the Automated Checks

The automated checks in this PoC showed varied success in assessing compliance (Table 4.6), illustrating both the potential and challenges of automating SSCS guideline verification. Straightforward checks, like SCVS 2.3 (SBOM unique identifier), were consistently and correctly verified with high conformance and confidence. This success, along with correct branch protection identification (SOK SG-016) for some projects, demonstrates the fundamental viability of the proposed attestation approach.

However, we also observed the limits of our simple methods. The SCVS 3.3 (CI Pipeline) check, for instance, failed to identify existing CI pipelines in all projects, reporting 0% conformance due to simplistic keyword matching. This highlights the difficulty in creating universal heuristics for complex, variable configurations. Furthermore, several PoC checks were intentionally platform-specific (GitHub, Node.js) to demonstrate the core attestation mechanics within a manageable scope; while this limits their general applicability, the positive results confirm the feasibility of the concept.

One way to obtain broader or more reliable security checks could be to utilize existing, well-known tools like OpenSSF Scorecard [13] or Legitify [14]. These tools assess general security health and best practices, rather than verifying if a project adheres to the exact framework rules we focused on in this study. However, some of the security ideas they check align with the principles in the software supply chain security guidelines. Using these tools could be another way to gather security information, either as an alternative to building custom checkers for specific guidelines or in conjunction with them to gain a more comprehensive understanding of a project's security. Their strong checks could provide good evidence for many security claims.

6
Conclusion

This thesis addressed software supply chain security through two main efforts. First, it conducted a comparative analysis of prominent security frameworks to clarify their foundational concepts and distinct roles. Second, it demonstrated that automating compliance checks for some of their guidelines is practically achievable. The overall goal was to bring clarity to this evolving and often fragmented field by providing both a structured comparison and a concrete example of how these guidelines can be implemented through automation.

A central outcome of the comparative analysis is the distinct niches occupied by each framework. The investigation revealed that, while numerous valuable resources exist, no single framework currently offers a "one-size-fits-all" solution that encompasses the entirety of best practices for securing the software supply chain. For instance, S2C2F provides targeted guidance for secure open-source consumption, SLSA offers unparalleled depth in build integrity and provenance, SCVS delivers structured component verification, and the SOK paper provides a broad range of safeguards. This specialization highlights the complexity of software supply chain security, suggesting that organizations may need to consult multiple frameworks to achieve a stronger overall security posture.

Among these specialized frameworks, ESF was identified as the most comprehensive due to its large number of guidelines and the wide range of security topics it covers. Its extensive recommendations address many developer-centric practices throughout the software development lifecycle. However, despite this breadth, the detailed focus of other frameworks on specific areas, like SLSA for build systems or S2C2F for dependency management, shows that ESF is most effective when used alongside these more targeted standards for stronger overall security.

Beyond theoretical analysis, this work also demonstrated the feasibility of an end-to-end automated compliance checking and attestation process. A PoC tool was developed to automatically assess adherence to a subset of decomposed software supply chain security guidelines for selected open-source projects. This involved generating verifiable compliance attestations embedded within an SBOM, incorporating granular evidence, distinct claims, and digital signatures to ensure integrity and authenticity. Additionally, a companion visualization tool was developed that parsed these enriched SBOMs, allowing for human review and cryptographic verification of the attestations. The results of the study indicate that while comprehensive automation for all guideline types remains a challenge, a specific subset can be ef-

fectively automated. This suggests that SBOMs can serve as a viable mechanism for communicating verifiable compliance.

6.1 Future Work

While this thesis has contributed to analyzing software supply chain security frameworks and demonstrating automated compliance attestation, future work could explore several promising directions to build on these findings:

- **Expanding and Refining Automated Verification:** Future efforts should focus on broadening the coverage of automated checkers to encompass a more extensive set of guidelines from the analyzed frameworks. This includes investigating the application of advanced AI techniques, such as LLMs, for verifying more qualitative or documentation-dependent guidelines, which were identified as challenging for traditional automation. Specifically, it would be highly valuable to explore how well LLM-based approaches perform in comparison to the static automation methods developed in this work. Such an investigation could reveal their potential to significantly enhance the versatility of compliance tools, perhaps even enabling them to interpret and verify new or evolving guidelines with minimal to no additional bespoke checker implementation.
- **Developing a Hybrid Attestation Model:** Given the limitations of full automation, a key future direction is the development of a hybrid attestation model. This involves user-friendly tools that allow experts to manually generate, review, and attest to compliance for guidelines unsuitable for automation. These attestations should follow the CycloneDX structure and use distinct digital signatures, enabling the visualization tool to clearly distinguish and verify both automated and manual assertions within a single SBOM.
- **Creating a Unified and Interactive Guideline Knowledge Base:** The comparative analysis highlighted the potential for a unified software supply chain security framework. Future work could adopt a modified version of our decomposition approach and apply it to guidelines from ESF, S2C2F, SCVS, SLSA, and SOK into a single, de-duplicated, and interactive web-based knowledge base. This platform could offer dynamic navigation, maturity modeling, links to attack vectors (from SOK), and tool recommendations, thereby significantly improving the accessibility and applicability of SSCS guidance for developers and organizations.

Further exploration of these topics will help build a more robust, transparent, and actionable approach to securing the software supply chain.

In conclusion, this work provides both a deeper understanding of the current software supply chain security guidance landscape and a practical pathway towards making compliance more automated, verifiable, and integrated into the software development lifecycle, with clear paths for future enhancements.

Bibliography

[1] "ENISA Threat Landscape 2024," European Union Agency for Cybersecurity (ENISA), Sep. 2024, Accessed: 2025-01-31. [Online]. Available: https://www.enisa.europa.eu/publications/enisa-threat-landscape-2024.

[2] E. O'Donoghue, A. M. Reinhold, and C. Izurieta, " Assessing Security Risks of Software Supply Chains Using Software Bill of Materials," in *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering - Companion (SANER-C)*, Los Alamitos, CA, USA: IEEE Computer Society, Mar. 2024, pp. 134–140. DOI: 10.1109/SANER-C62648.2024.00023. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SANER-C62648.2024.00023.

[3] S. Konecka and Z. Bentyn, "Cyberattacks as threats in supply chains," *European Research Studies Journal*, vol. XXVII, no. 3, pp. 778–796, 2024. DOI: 10.35808/ersj/3467. [Online]. Available: https://ersj.eu/journal/3467.

[4] Sonatype, *The 8th annual state of the software supply chain report*, Technical Report, Accessed: 2025-01-31, 2023. [Online]. Available: https://www.sonatype.com/hubfs/1-2023%20New%20Site%20Assets/SSCR/8th-Annual-SSCR-digital-0206%20update.pdf.

[5] R. Alkhadra, J. Abuzaid, M. AlShammari, and N. Mohammad, "Solar winds hack: In-depth analysis and countermeasures," in *2021 12th International Conference on Computing Communication and Networking Technologies (ICCCNT)*, 2021, pp. 1–7. DOI: 10.1109/ICCCNT51525.2021.9579611.

[6] R. Hiesgen, M. Nawrocki, T. C. Schmidt, and M. Wählisch, *The race to the vulnerable: Measuring the log4j shell incident*, 2022. arXiv: 2205.02544 [cs.CR]. [Online]. Available: https://arxiv.org/abs/2205.02544.

[7] O. Foundation, *Software component verification standard (scvs)*, Accessed: 2025-03-05, 2023. [Online]. Available: https://scvs.owasp.org/.

[8] Google and OpenSSF, *Supply-chain levels for software artifacts (slsa)*, Accessed: 2025-03-05, 2023. [Online]. Available: https://slsa.dev/.

[9] O. S. S. F. (OpenSSF), *Secure supply chain consumption framework (s2c2f)*, Accessed: 2025-03-05, 2023. [Online]. Available: https://github.com/ossf/s2c2f/blob/main/specification/framework.md.

[10] Cybersecurity and I. S. A. (ESF), *Securing the software supply chain: Recommended practices for developers*, Accessed: 2025-03-05, 2022. [Online]. Available: https://www.cisa.gov/sites/default/files/publications/ESF_SECURING_THE_SOFTWARE_SUPPLY_CHAIN_DEVELOPERS.PDF.

[11] C. Ladisa, E. Mariconti, and L. Cavallaro, "Sok: Taxonomy of attacks on open-source software supply chains," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, Accessed: 2025-03-05, 2023. [Online]. Available: https://oaklandsok.github.io/papers/ladisa2023.pdf.

[12] P. Ladisa, S. E. Ponta, A. Sabetta, M. Martinez, and O. Barais, "Journey to the center of software supply chain attacks," *arXiv preprint arXiv:2304.05200*, 2023. [Online]. Available: https://arxiv.org/abs/2304.05200.

[13] Open Source Security Foundation, *OpenSSF Scorecard: Security health metrics for Open Source*, https://github.com/ossf/scorecard, Accessed: 2025-05-19, 2025.

[14] Legit Labs, *Legitify: Detect and remediate misconfigurations and security risks across GitHub and GitLab assets*, https://github.com/Legit-Labs/legitify, Accessed: 2025-05-19, 2025.

[15] OWASP Foundation, *Authoritative Guide to Attestations*, https://cyclonedx.org/guides/OWASP_CycloneDX-Authoritative-Guide-to-Attestations-en.pdf, Accessed: 2025-05-19, 2024.

[16] European Parliament and Council of the European Union. "Regulation (eu) 2024/2847 of the european parliament and of the council of 23 october 2024 on horizontal cybersecurity requirements for products with digital elements and amending regulations (eu) no 168/2013 and (eu) no 2019/1020 and directive (eu) 2020/1828 (cyber resilience act)." Accessed: 2025-05-14. (Nov. 2024), [Online]. Available: https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX%3A32024R2847.

[17] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *NAACL*, 2019. [Online]. Available: https://aclanthology.org/N19-1423.

[18] V. Orbinati, D. E. Andreychenko, and M. Alfjord, "Automated mapping of security advisories to mitre att&ck," in *RAID*, 2020.

[19] Z.-X. Xie, C.-P. Nagy, and D. Basin, "Text2policy: Automated extraction of access control policies from natural language text," in *ESORICS Workshops*, 2018.

[20] H. Ding, H. Gu, and P. Cao, "Research on an event extraction framework based on two-step prompt learning for chinese policy," *Applied Sciences*, vol. 15, no. 6, p. 3378, 2025. DOI: 10.3390/app15063378. [Online]. Available: https://www.mdpi.com/2076-3417/15/6/3378.

[21] M. Wanner, S. Ebner, Z. Jiang, M. Dredze, and B. Van Durme, *A closer look at claim decomposition*, arXiv:2403.11903v1, 2024. arXiv: 2403.11903 [cs.CL].

[22] S. Torres-Arias, H. Afzali, T. K. Kuppusamy, R. Curtmola, and J. Cappos, "In-toto: Providing farm-to-table guarantees for bits and bytes," in *Proceedings of the 28th USENIX Security Symposium*, Santa Clara, CA, USA: USENIX Association, 2019, pp. 1393–1410. [Online]. Available: https://www.usenix.org/system/files/sec19-torres-arias.pdf.

[23] Sigstore Project, *Cosign: Code Signing and Transparency for Containers and Binaries*, https://github.com/sigstore/cosign, Accessed: 2025-05-19, 2025.

[24] MinderSec, *Minder: Software Supply Chain Security Platform*, https://github.com/mindersec/minder, Accessed: 2025-05-19, 2025.
[25] National Institute of Standards and Technology, *Open Security Controls Assessment Language (OSCAL)*, https://pages.nist.gov/OSCAL/about/, Accessed: 2025-05-19, 2025.
[26] National Institute of Standards and Technology (NIST), "Cybersecurity Supply Chain Risk Management Practices for Systems and Software (SP 800-161 Revision 1)," NIST, Tech. Rep., 2023. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-161r1-upd1.pdf.
[27] B. M. Reichert and R. R. Obelheiro, "Software supply chain security: A systematic literature review," *International Journal of Computers and Applications*, vol. 46, no. 10, pp. 853–867, 2024, Accessed: 2025-01-30. DOI: 10.1080/1206212X.2024.2390978. [Online]. Available: https://www.tandfonline.com/doi/full/10.1080/1206212X.2024.2390978.
[28] B. Hammi and S. Zeadally, "Software supply-chain security: Issues and countermeasures," *Computer*, vol. 56, no. 7, pp. 54–66, 2023. DOI: 10.1109/MC.2023.3273491.
[29] B. Gokkaya, L. Aniello, and B. Halak, "Software supply chain: Review of attacks, risk assessment strategies and security controls," *ArXiv*, vol. abs/2305.14157, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:258841711.
[30] M. S. Melara and M. Bowman, "What is software supply chain security?" *CoRR*, vol. abs/2209.04006, 2022. DOI: 10.48550/arXiv.2209.04006. [Online]. Available: https://arxiv.org/abs/2209.04006.
[31] C. Okafor, T. R. Schorlemmer, S. Torres-Arias, and J. C. Davis, "Sok: Analysis of software supply chain security by establishing secure design properties," in *Proceedings of the ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses (SCORED 22)*, 2022, pp. 15–24. DOI: 10.1145/3548606.3563443. [Online]. Available: https://doi.org/10.1145/3548606.3563443.
[32] Y. Shen, X. Gao, H. Sun, and Y. Guo, "Understanding vulnerabilities in software supply chains," *Empirical Software Engineering*, 2024. DOI: 10.1007/s10664-024-10581-2. [Online]. Available: https://doi.org/10.1007/s10664-024-10581-2.
[33] K. G. Kalu, T. Singla, C. Okafor, S. Torres-Arias, and J. C. Davis, *An industry interview study of software signing for supply chain security*, arXiv preprint, 2024. DOI: 10.48550/arXiv.2406.08198. [Online]. Available: https://arxiv.org/abs/2406.08198.
[34] W. Enck, L. Williams, G. Benedetti, and S. Hamer, "Research directions in software supply chain security," *ACM Transactions on Software Engineering and Methodology*, 2025, to appear. DOI: 10.1145/3714464. [Online]. Available: https://doi.org/10.1145/3714464.
[35] Cybersecurity and Infrastructure Security Agency (CISA), *Securing the software supply chain: Recommended practices for suppliers*, 2024. [Online]. Avail-

able: https://www.cisa.gov/sites/default/files/2024-08/SECURING_THE_SOFTWARE_SUPPLY_CHAIN_SUPPLIERS_508.pdf.

[36] A. Ahmed and A. Abdullah, "Enhancing software supply chain resilience: Strategy for mitigating software supply chain security risks and ensuring security continuity in development lifecycle," *International Journal on Soft Computing*, vol. 15, no. 1/2, pp. 01–18, May 2024, ISSN: 2229-7103. DOI: 10.5121/ijsc.2024.15201. [Online]. Available: http://dx.doi.org/10.5121/ijsc.2024.15201.

[37] National Institute of Standards and Technology, *Glossary software bill of materials*, https://csrc.nist.gov/glossary/term/software_bill_of_materials, Accessed: 2025-05-29.

[38] B. Xia, T. Bi, Z. Xing, Q. Lu, and L. Zhu, "An empirical study on software bill of materials: Where we stand and the road ahead," *arXiv preprint arXiv:2301.05362*, 2023. [Online]. Available: https://arxiv.org/abs/2301.05362.

[39] S. Workgroup, *Spdx specification*, ISO/IEC 5962:2021, 2021. [Online]. Available: https://spdx.github.io/spdx-spec/.

[40] OWASP Foundation and Ecma International Technical Committee for Software & System Transparency (TC54). "CycloneDX: The international standard for bill of materials (ecma-424)." (2025), [Online]. Available: https://cyclonedx.org/ (visited on 05/28/2025).

[41] Anchore, Inc., *Syft: CLI tool and library for generating a Software Bill of Materials from container images and filesystems*, https://github.com/anchore/syft, Accessed: 2025-05-19, 2025.

[42] Aqua Security, *Trivy: Comprehensive and versatile security scanner*, https://github.com/aquasecurity/trivy, Accessed: 2025-05-19, 2025.

[43] CycloneDX Project, *cdxgen: Multi-language SBOM Generator*, https://github.com/CycloneDX/cdxgen, Accessed: 2025-05-19, 2025.

[44] Chainloop Inc., *Chainloop: Evidence Store and Policy Engine for Software Supply Chain Attestations*, https://github.com/chainloop-dev/chainloop, Accessed: 2025-05-19, 2025.

[45] OWASP Foundation, *CycloneDX Use Case: Authenticity Verification*, https://cyclonedx.org/use-cases/authenticity-verification/, Accessed: 2025-05-19, 2025.

[46] K. Heard-Rising, *Cyclonedx v1.6 introduces support for attestations of compliance with any standard*, https://owasp.org/blog/2023/12/06/CycloneDX-attestations, Accessed: 2025-05-19, 2023.

[47] J. Gatto, O. Sharif, P. Seegmiller, P. Bohlman, and S. M. Preum, "Text encoders lack knowledge: Leveraging generative llms for domain-specific semantic textual similarity," *arXiv preprint arXiv:2309.06541*, Sep. 2023, Under review GEM@EMNLP-2023, 12 pages. DOI: 10.48550/arXiv.2309.06541. [Online]. Available: https://arxiv.org/abs/2309.06541.

[48] D. Chandrasekaran and V. Mago, "Evolution of semantic similarity - A survey," *CoRR*, vol. abs/2004.13820, 2020. arXiv: 2004.13820. [Online]. Available: https://arxiv.org/abs/2004.13820.

[49] K. Song, X. Tan, T. Qin, J. Lu, and T. Liu, “Mpnet: Masked and permuted pre-training for language understanding,” *CoRR*, vol. abs/2004.09297, 2020. arXiv: 2004.09297. [Online]. Available: https://arxiv.org/abs/2004.09297.

[50] S. Transformers, *Pretrained models*, https://www.sbert.net/docs/sentence_transformer/pretrained_models.html, Accessed: 2025-03-31.

[51] P. Mayring, “Qualitative content analysis,” *Forum: Qualitative Social Research*, vol. 1, no. 2, Art. 20, 2000. DOI: 10.17169/fqs-1.2.1089. [Online]. Available: https://www.qualitative-research.net/index.php/fqs/article/view/1089.

[52] M. Corporation, *MITRE ATT&CK Framework*, https://attack.mitre.org/, Accessed: 2025-04-01.

[53] European Cyber Security Organisation, *Technical Paper: Software Supply Chain Security*, https://ecs-org.eu/?publications=technical-paper-software-supply-chain-security, Accessed: 2024-12-06.

[54] J. Nielsen and A. Kaley, *Pdf: Still unfit for human consumption, 20 years later*, https://www.nngroup.com/articles/pdf-unfit-for-human-consumption/, Accessed: 31 May 2025, Aug. 2020.