Evaluation and Improvement of C Based Dependency Ecosystems in SCA Tool

BACHELOR THESIS

Junzhe Wang

Submitted on 19 September 2025

Friedrich-Alexander-Universität Erlangen-Nürnberg
Faculty of Engineering, Department Computer Science
Professorship for Open Source Software

Supervisor:
Martin Wagner
Prof. Dr. Dirk Riehle, M.B.A.

Friedrich-Alexander-Universität
Faculty of Engineering

Declaration of Originality

Erlangen, 19 September 2025

License

Erlangen, 19 September 2025

Abstract

Open Source Software (OSS) is widely adopted, but introduces security and license compliance risks that must be managed. Software Composition Analysis (SCA) tools address these challenges by identifying dependencies, generating Software Bill of Materials (SBOM), and detecting vulnerabilities. Although *SCA Tool* already supported ecosystems such as `npm`, it lacked support for C-based (e.g., C, C++) projects, which are central to many critical domains.

This thesis extends *SCA Tool* with integration for the `vcpkg` package manager. Four new components were implemented: the `VcpkgExtractor` for parsing manifests, the `VcpkgCommand` for interacting with the Command-Line Interface (CLI) and the `VcpkgResolver` for metadata enrichment. Core services were updated to incorporate these seamlessly.

The evaluation shows that the integration successfully extracts and enriches the declared and transitive dependencies, producing SBOMs consistent with the tool data model. Although limited to the official registry, this work provides a foundation for broader ecosystem support.

Contents

List of Figures

List of Tables

Acronyms

OSS	Open Source Software
API	Application Programming Interface
TTM	Time-To-Market
ORT	Open-Source Software Review Toolkit
SBOM	Software Bill of Materials
CVE	Common Vulnerabilities and Exposures
SPDX	Software Package Data Exchange
URL	Uniform Resource Locator
HPC	High-Performance Computing
UI	User Interface
CLI	Command-Line Interface
SCA	Software Composition Analysis
EU	European Union
BOM	Bill of Materials

1 Introduction

OSS has become the foundation of modern software development, enabling rapid innovation, accelerating Time-To-Market (TTM) and significantly reducing development costs. Organizations in all industries are now relying on OSS components to build scalable and feature-rich applications. However, the widespread adoption of OSS also introduces substantial challenges, particularly in the areas of license compliance, security, and governance. The diversity of OSS licenses, each with its own obligations and restrictions, creates legal and operational risks if not managed properly. Additionally, the use of third-party components increases the potential attack surface, making it critical to identify and address vulnerabilities in dependencies. To address these complexities, *SCA Tool* has become an essential solution. *SCA Tool* automates the detection and inventory of OSS components, analyzes their associated licenses, and assesses security vulnerabilities, allowing organizations to maintain compliance, mitigate risks, and ensure the integrity of their software supply chain.

SCA Tool, a React application based on TypeScript with a Spring Java back-end, is designed to streamline OSS license compliance by scanning project lockfiles - such as *package-lock.json* or *npm-shrinkwrap.json* - to identify all associated software components. Currently, *SCA Tool* supports the `npm` package ecosystem, primarily for JavaScript and TypeScript projects, using the Open-Source Software Review Toolkit (ORT) and ScanCode for automated license detection and dependency analysis. This integration enables efficient and accurate identification of third-party components and their associated licenses, particularly for languages and frameworks with well-established package management systems such as `npm`, `Maven`, `Bazel`, and others. However, despite its effectiveness in these domains, the support for C-based (C, C++) dependency ecosystems in *SCA Tool* remains limited. C and C++ are widely used in critical infrastructure, embedded systems, and legacy applications, where dependency management is often less standardized and more complex. As a result, expanding the compatibility of *SCA Tool* to handle C and C++ projects robustly is essential to achieve complete license compliance, mitigate legal and security risks, and support organizations that rely on these foundational technologies.

This thesis aims to evaluate the current integration of C-based dependency analysis within *SCA Tool* and propose improvements. Enhancing support for C ecosystems will not only broaden the applicability of the tool, but also ensure consistency across diverse software stacks.

2 Literature Review

To effectively explore SCA, the following chapter aids in establishing a solid foundation in key concepts such as SBOM, open source governance, and license compliance, which together provide the context for managing software risks and ensuring regulatory alignment.

2.1 SBOM

The Software Bill of Materials (SBOM) has become the cornerstone of modern software supply chain security. An SBOM lists all components, dependencies, and their versions within a software product, enabling organizations to assess vulnerabilities and license risks. After high-profile attacks such as SolarWinds, the US government mandated the adoption of SBOM through Executive Order 14028 (CISA, 2022; NTIA, 2021).

To standardize SBOMs, formats such as Software Package Data Exchange (SPDX) (L. Foundation, 2025) and CycloneDX (O. Foundation, 2025) have gained traction. SPDX is widely used in open source communities, while CycloneDX is popular in industry for security-focused metadata. Comparative studies show that the choice of format affects compliance and interoperability (Nocera et al., 2025; Tobar et al., 2025).

Research indicates a steady growth in the adoption of SBOMs. Nocera et al. (2025) highlight its increasing use in GitHub-hosted projects, while Tobar et al. (2025) emphasize ongoing harmonization efforts. Chaora and Camp (2025) collected SBOMs from over 100,000 repositories, illustrating real-world adoption patterns. Putta and Pendyala (2025) found that continuous SBOM generation in CI/CD pipelines leads to faster vulnerability detection than static build-time approaches.

Recent works explore tool performance and accuracy. Garcia et al. (2025) compare tools such as Syft, Trivy, and SPDX-Tool, showing trade-offs between coverage and speed. Rabbi et al. (2025) analyze Rust tools SBOM, revealing inconsistencies in the CycloneDX vs. SPDX output. Anthony et al. (2025) underline the importance of linking SBOM data with Common Vulnerabilities and Exposures (CVE)s. These studies show both promise and challenge: SBOMs improve transparency, but interoperability and real-time accuracy remain open problems.

2.2 Open Source Governance

Open source governance refers to the policies and practices that guide responsible adoption of OSS, balancing innovation with risk mitigation. It encompasses compliance management, security oversight, and sustainability planning.

Corti et al. (2025) examine how organizations adapt to the NIS2 Directive and the European Union (EU) Cyber Resilience Act, finding that governance requires both legal and technical compliance. Humphrey (2025) shows that start-ups in regulated industries such as healthcare need 'guardrails' to manage OSS risks. Yahattaa (2025) models key management weaknesses and their impact on organizational security.

Governance is increasingly based on tools and metrics. The OpenSSF Scorecards project evaluates OSS projects by measuring security health indicators (OpenSSF, 2025). Scott and Sims (2025) argue that governance is not purely technical: community engagement and contributor diversity also influence project sustainability.

Thus, governance is shifting from compliance-only to a holistic framework in which organizations combine technical controls, legal processes, and community participation.

2.3 License Compliance

Open-source licenses (e.g., GPL, MIT, Apache) define how software may be used and redistributed. Ensuring license compliance is crucial to avoid legal, reputational, and operational risks. However, research reveals frequent inconsistencies: Barcomb et al. (2023) show mismatched license declarations across GitHub projects, while Fendt and Jaeger (2019) highlight systemic challenges.

Specialized SCA tools automate compliance. Tools such as ScanCode, FOSSology, and ORT are widely used for automated detection of license obligations (Trenz et al., 2023; Wagner, 2023). The OpenChain standard provides a recognized framework for OSS compliance management and has been adopted internationally (L. Foundation, 2020).

Systematic studies confirm the limitations of the tool. Liu et al. (2025) find that license conflicts persist even with SPDX identifiers. Antelmi et al. (2024) analyze license usage on a scale, highlighting classification errors. Serafini and

Zacchiroli (2022) recommend using multiple tools (e.g., ScanCode + FOSSology) for accuracy.

In practice, compliance workflows now integrate into CI/CD pipelines. Fendt and Jaeger (2019) argue that coupling building systems with compliance tools provides both legal and technical guarantees. This evolution underscores compliance as a continuous, automated process.

2.4 Common Build Tools & Package Managers in C-Based (C / C++) Projects

C-based (C, C++) projects differ from ecosystems such as `npm` or `Gradle` in that they do not rely on a single integrated system for dependency management and build orchestration. Instead, developers typically combine separate tools: build systems to compile and link code, and package managers to retrieve and track external libraries. This fragmentation complicates reproducibility, license compliance, and vulnerability management. For the purposes of this thesis, it is important to understand the landscape of both build systems and package managers, as they define the entry points for extending *SCA Tool* into C-based ecosystems. The following subsections provide an overview of these two categories.

2.4.1 Build Systems in C / C++ Projects

Build systems orchestrate compilation, linking, and testing in C and C++ projects. Make, created in the 1970s, remains widely used, but suffers from portability issues. `CMake`, introduced in 2000, became the de facto standard for large-scale projects thanks to cross-platform support and modularity (Petrillo, 2025).

`Ninja` focuses on speed, making it popular for CI/CD environments where incremental builds matter (Garcia et al., 2025). `Bazel`, developed by Google, enables hermetic builds and advanced caching, which are increasingly being used in large enterprise and research contexts. The `Meson` simplifies the build configuration with a Python-like syntax, appealing to smaller teams.

Recent research emphasizes that building systems are now critical to security. They can automatically generate SBOMs, enforce secure compiler flags, and integrate with license scanners (Garcia et al., 2025; Nocera et al., 2025). Thus, build systems are evolving from compilation orchestrators to compliance enforcers.

2.4.2 Package Management for C / C++ Dependencies

Historically, C/C++ lacked standardized package management, which led developers to manually manage dependencies. This caused 'dependency hell', version conflicts, and compliance blind spots.

Modern tools have transformed this landscape:

- **Conan**: supports reproducible builds and enterprise workflows, integrating with `CMake` and `Meson` (Serafini & Zacchiroli, 2022).
- **vcpkg**: developed by Microsoft, provides curated recipes for thousands of libraries and integrates tightly with Visual Studio (Microsoft, 2025).
- **Spack**: designed for High-Performance Computing (HPC), manages complex dependency trees between architectures, crucial in scientific computing (Antelmi et al., 2024).

Industry reports show convergence: `Conan` and `vcpkg` now include license metadata, while `Spack` integrates with HPC compliance frameworks (Corti et al., 2025). Tools such as OSS Index and VulnerableCode further connect these package managers to vulnerability databases (Rabbi et al., 2025).

The future trend is security-aware package management, where dependency resolution, vulnerability scanning, and license compliance are unified into one automated workflow.

2.5 SBOM Ecosystem and the Situation in Mainland China

In recent years, SBOM has gained traction worldwide as a means of improving transparency, traceability, and resilience in software supply chains. China has increasingly emphasized the adoption of SBOM due to the increasing threat of supply chain vulnerabilities and the geopolitical importance of software security. Several initiatives, such as government-led frameworks and industry-driven practices SBOM, have been proposed to improve visibility into open-source and proprietary components used in critical infrastructure (W. Wu et al., 2023).

Chinese researchers have explored SBOM applications in IoT and embedded system security, stressing the importance of standardized formats to improve vulnerability detection and incident response (W. Wu et al., 2023). Studies on open-source ecosystems, such as the RISC-V ecosystem, also highlight the role of SBOM in enabling transparency between emerging technology stacks in China (Y. Wu et al., 2024). At the same time, empirical research on SBOM adoption indicates that China faces similar challenges to other countries, such as integration with legacy systems and the lack of uniform regulatory enforcement (Xia et al., 2023).

Prototypes such as LiPSBOMaker demonstrate practical solutions for generating SBOMs in complex Linux distributions, signaling progress in tool development within Chinese academia and industry (Qiu et al., 2025). Meanwhile, ongoing work focuses on risk assessment and security assurance in open source software supply chains, aligning SBOM efforts with global practices while addressing domestic policy concerns (Feng et al., 2025). These efforts reflect a growing recognition in China that SBOM adoption is not only a technical necessity, but also a strategic component of digital sovereignty and cybersecurity.

3 Preliminary Assessment of Existing C Based Analysis within SCA Tool

At present, *SCA Tool* does not include any built-in functionality for analyzing C-based (C / C++) dependency systems. Instead, it fully relies on the capabilities of ORT for processing C-based projects.

Within ORT, support for C-based dependency analysis is limited to `Conan`-specific integration. This means that projects using other widely adopted C / C++ package managers — such as `vcpkg`, `Spack`, or `Hunter` — or those depending solely on the build system (e.g., `CMake`, `Meson`, `Makefile`) are not natively supported. Consequently, dependency extraction, metadata resolution, and SBOM generation for these ecosystems cannot be automated without additional tooling or custom analyzers.

For example, a `Conan`-managed project can be analyzed by ORT using its dedicated analyzer:

```kotlin
@OrtPlugin(
    displayName = "Conan",
    description = "The Conan package manager for C / C++.",
    factory = PackageManagerFactory::class
)
@Suppress("TooManyFunctions")
class Conan(
    override val descriptor: PluginDescriptor = ConanFactory.descriptor,
    private val config: ConanConfig
) : PackageManager("Conan") {
    companion object {
        internal val DUMMY_COMPILER_SETTINGS = arrayOf(
            "-s", "compiler=gcc",
            "-s", "compiler.libcxx=libstdc++",
            "-s", "compiler.version=11.1"
        )

        internal const val SCOPE_NAME_DEPENDENCIES = "requires"
        internal const val SCOPE_NAME_DEV_DEPENDENCIES = "build_requires"
    }

    internal val command by lazy { ConanCommand(config.useConan2) }

    override val globsForDefinitionFiles = listOf("conanfile*.txt", "conanfile*.py")

    private val handler by lazy {
        if (command.getVersion().startsWith("1.")) {
            ConanV1Handler(this)
        } else {
            ConanV2Handler(this)
        }
    }
    ...
}
```

Listing 3.1: Conan Handler Code Snippet from ORT

However, a project relying on `vcpkg` alone does not benefit from equivalent out-of-the-box support. Instead, *SCA Tool* currently treats it as a generic source tree without dependency resolution.

```kotlin
@Service
class ExtractionService {
    private val extractors: List<Extractor> = listOf(
       NpmExtractor())
    fun resolveBom(input: Path): ExtractionResult {
        val extractors = extractors.mapNotNull { it.extract(
           input) }
        if (extractors.isNotEmpty()) {
        return extractors.reduce(ExtractionResult::merge)
        }
        return ExtractionResult(emptyMap(), emptyMap(),
           emptyMap())
    }
}
```

Listing 3.2: Current ExtractionService Snippet from SCA Tool

As a result, C-based analysis in *SCA Tool* is constrained both by the absence of native features and by the restricted scope of ORT's current C / C++ support, leaving substantial gaps in license compliance and dependency transparency for the majority of C-based projects.

3.1 Scope and Limitations of Current Implementation

The scope of C-based (C, C++) dependency analysis in the current *SCA Tool* is fully defined by the capabilities of ORT, as no native support has been implemented within the tool itself. As a result, the operational scope is narrow and is centered on `Conan`-specific integration.

3.1.1 Scope of Current Implementation

- Supported Package Manager: `Conan` is the only C/C++ package manager with native analysis support through ORT.
- Dependency Resolution: Capable of parsing `conanfile.txt` or `conanfile.py` manifests to extract dependency names, versions, and sources.
- License Detection: Leverage ORT's scanning capabilities to identify licenses for resolved `Conan` dependencies and source files.
- Vulnerability Detection: Potential to map `Conan`-resolved dependencies to known CVEs using the ORT vulnerability processing pipeline.

- SBOM Generation: Limited to dependencies detected via `Conan`; outputs in standard ORT supported formats (SPDX, CycloneDX).

3.1.2 Limitations of Current Implementation

1. **Lack of Multi-Manager Support**
    - No native handling for `vcpkg`, `Spack`, `Hunter`, or `CPM`.
    - Projects using only build systems (e.g., `CMake`, `Meson`) without `Conan` are not supported for automated dependency resolution.
2. **Incomplete SBOM Coverage**
    - SBOM generation is partial and omits dependencies not managed by `Conan`.
    - Can not produce a full component inventory for non-`Conan` projects.

In summary, the scope is essentially `Conan`-based C/C++ projects, and the limitations prevent *SCA Tool* from delivering comprehensive license compliance, vulnerability detection, and SBOM generation across the broader C and C++ ecosystem.

4 Goals

Building on the preliminary assessment in Chapter 3, this chapter outlines the main goals of the thesis. Although earlier chapters established the challenges of limited C-based support in *SCA Tool*, the following sections define the objectives that guide the extension of the tool. These goals serve as a link between the identified gaps and the concrete requirements of Chapter 5.

Specifically, the focus is twofold: First, to broaden the applicability of *SCA Tool* beyond `npm`-centric projects and into C/C++ ecosystems. Second, to justify the selection of `vcpkg` as the initial package manager for integration. Together, these goals provide a foundation for the architectural and implementation work described in the following chapters.

4.1 Extending the Applicability of SCA Tool

A central goal of this thesis is to broaden the applicability of *SCA Tool* beyond its original focus on `npm`-based projects. Although `npm` integration has proven to be effective, it only covers a fraction of the ecosystems used in practice. Many critical domains, including embedded systems, HPC, and infrastructure software, are heavily based on C and C++ dependencies. These domains are often under-represented in existing SCA tools despite their importance in safety-critical and enterprise contexts.

By integrating support for `vcpkg`, *SCA Tool* is extended to these usages, increasing its relevance in a wider spectrum of software projects. This goal goes beyond feature parity with `npm`: it aims to position *SCA Tool* as a genuinely multi-ecosystem analyzer, capable of addressing both modern web technologies and foundational system-level software. In doing so, the tool becomes more useful for organizations seeking a unified compliance and security solution across diverse technology stacks.

4.2 Why vcpkg for SCA Tool Implementation

A fundamental objective of this thesis is to extend *SCA Tool* with reliable support for C-based ecosystems. The selection of an appropriate package manager is critical to the achievement of this goal, as it determines the feasibility of integration, the quality of dependency resolution, and the precision of the generated SBOMs. After surveying existing options such as `Conan`, `Spack`, `Hunter`, and others, `vcpkg` was chosen as the primary target for integration. This decision is guided by the overarching goals of reproducibility, simplicity of integration, and traceability in software supply chains.

The rationale for this choice can be structured as follows:

- **Cross-Platform Coverage:** *SCA Tool* aims to support heterogeneous development environments. `vcpkg` is natively supported on Windows, macOS, and Linux, making it an ideal candidate to ensure broad applicability across industry contexts.
- **Tight `CMake` Integration:** As `CMake` has become a very popular build standard in modern C-based projects (C, C++), the seamless interoperability of `vcpkg` with `CMake` directly contributes to one of the thesis objectives: lowering the barrier to adoption and minimizing the required changes in existing build workflows (Gygi, 2021; Microsoft, 2025; Świdziński, 2024).
- **Rich Ecosystem and Port Availability:** With more than 2,000 prebuilt packages available in its central registry, `vcpkg` ensures that a wide variety of projects can be supported out of the box (Microsoft, 2025). This accelerates prototyping, reduces maintenance overhead, and guarantees reproducibility in dependency management. However, many C and C++ libraries are not yet covered by `vcpkg`. In comparison, `Spack` supports nearly 8,500 HPC-focused packages (Lawrence Livermore National Laboratory, 2025). Although the number of `vcpkg` ports has steadily increased from approximately 200 in 2017 to more than 2,200 in 2025 (Popa, 2025), its coverage remains narrower than that of competing ecosystems. This illustrates both the strong growth trend of `vcpkg` and the ongoing need for supplementary solutions when projects depend on libraries outside of its registry.
- **Community and Industry Adoption:** Backed by Microsoft, `vcpkg` has seen rapid growth and adoption in both open source and enterprise settings. This strong ecosystem support improves long-term sustainability and aligns with the thesis goal of selecting a solution that will remain relevant in practice.
- **Complementarity with Existing ORT Support:** The ORT, on which

> *SCA Tool* is partially based, already provides native support for `Conan`. In contrast, there is currently no support for `vcpkg` within ORT or within *SCA Tool*. Using `vcpkg`, this thesis directly addresses an existing capability gap and provides genuine added value rather than duplicate existing functionality.

In contrast, while `Conan` provides greater flexibility and fine-grained binary management, its decentralized registry model and steeper learning curve make it less aligned with the immediate goals of this work, which emphasize simplicity, standardization, and reproducibility. Moreover, the ORT already includes native support for `Conan`, which means extending *SCA Tool* with `vcpkg` addresses an existing capability gap rather than duplicating functionality. Similarly, `Spack` is tailored for HPC scenarios, making it less suitable for the general-purpose cross-platform objectives pursued here.

In conclusion, `vcpkg` was selected not only for its technical capabilities, but also because it directly supports the strategic goals of this thesis: ensuring compliance, improving transparency through SBOMs, and allowing seamless extension of *SCA Tool* to the C-based ecosystem. This choice provides a pragmatic yet future-oriented foundation for demonstrating how *SCA Tool* can evolve into a multi-ecosystem analyzer.

5 Requirements

The purpose of this section is to define the requirements for extending *SCA Tool* to support C-based dependency ecosystems, with a primary focus on the `vcpkg` package manager. These requirements are derived from the gap analysis in Chapter 3 and are intended to ensure that the new functionality achieves parity with the existing `npm` integration in terms of automation, accuracy, and output quality. The requirements are grouped into *functional* and *non-functional* categories to clearly distinguish between the capabilities the system must provide and the quality attributes it must uphold.

5.1 Functional Requirements

1. **Manifest Parsing:** The system shall be able to parse the `vcpkg.json` to extract all declared dependencies.
2. **Dependency Resolution:** The system shall resolve direct and transitive dependencies by invoking the `vcpkg` CLI (`install --dry-run`, `depend-info`) against the registry.

   Note: Since `vcpkg` does not natively provide lock files, the resolution process may lead to slightly different results depending on the point in time when the analysis is executed. This limitation is inherent to the ecosystem when interpreting results.
3. **Analyzer Output for SBOM Export:** The analyzer system shall not generate SBOM files itself. Instead, it shall emit a completed internal *SCA Tool* data model `AnalyzerResult` that is accepted as-is by the monolith SBOM module. The analyzer output shall include, per component: package URL (PURL) or equivalent identifier, resolved version, dependency relationships, source location (e.g., registry path/URL), and checksums where available, plus normalized license information. When the monolith SBOM module consumes this `AnalyzerResult`, it can serialize valid SPDX or CycloneDX documents without additional transformations or schema validation errors.

4. **License Detection:** The system shall identify licenses for all `vcpkg` packages using a multi-step approach:
    - Use license metadata available in the `vcpkg` registry.
    - Normalize all detected license texts into SPDX-compliant license expressions.
5. **Integration with Existing Architecture:** The new `vcpkg` analysis module shall integrate seamlessly into the current *SCA Tool* architecture:
    - A dedicated `VcpkgExtractor` shall be implemented to handle manifest parsing and dependency graph extraction.
    - A `VcpkgResolver` shall enrich extracted packages with metadata (version, license, checksum, maintainers, source Uniform Resource Locator (URL)).
    - The existing `ExtractionService` and `MetadataService` shall be extended to support `vcpkg` along with other ecosystems.
    - Integration must support projects that combine multiple ecosystems (e.g., `npm` + `vcpkg`) without conflicts.

5.2 Non-Functional Requirements

1. **Performance:** The analysis must be completed within acceptable time (e.g., less than 5 minutes for a project with up to 200 dependencies).
2. **Accuracy:** The dependency resolution and SBOM generation processes shall achieve an accuracy rate of at least 95% for correctly identified dependencies and licenses.
3. **Cross-Platform Compatibility:** The `vcpkg` integration shall operate on Windows, Linux, and macOS without requiring platform-specific modifications.
4. **Standards Compliance:** The `vcpkg` integration components shall produce complete and correctly structured dependency and metadata objects that conform to the internal *SCA Tool* data model, such that the `AnalyzerResult` can be successfully serialized into valid SPDX and CycloneDX documents by the downstream SBOM export modules without schema validation errors.
5. **Maintainability:** The implementation shall follow the existing *SCA Tool* coding standards and be modular to support future integrations of the package manager.

6 Architecture

This chapter describes the architecture of *SCA Tool* and the proposed extension to support the `vcpkg` package manager for C/C++ dependency analysis. The current *SCA Tool* is built using a modular architecture that separates the front-end, back-end, and analysis layers. The new `vcpkg` integration is designed to align with existing design patterns, ensuring minimal disruption to the existing system while adding full support for `vcpkg`-based workflows.

6.1 Existing Architecture of SCA Tool

SCA Tool consists of four main layers:

- **Frontend (React + TypeScript):** Provides the User Interface (UI) to upload projects, initiate analysis workflows, and present the resulting reports in an interactive and accessible way.
- **Backend (Java Spring + Kotlin):** Acts as the orchestration layer, managing the end-to-end analysis process. Coordinate tasks between components, invoke the ORT or customized analyzer for dependency scanning, process intermediate results, and persist the final output to the storage system, performing vulnerability analysis and generating SBOMs in standardized formats.
- **Analysis Layer (ORT-based):** Implements the core logic for interpreting project manifests or lockfiles, resolving direct and transitive dependencies, and detecting licenses.
- **Scanner Layer (ScanCode Toolkit):** Specializes in the scanning of source codes for license texts, copyright statements and related metadata. It operates as a dedicated subsystem within the analysis pipeline, providing accurate and detailed licensing information to the Analysis Layer.

Within the analysis layer, key components include:

- **ExtractionService:** Parses manifest and build files to extract dependency information.
- **MetadataService:** Resolves metadata such as version, source URL, and checksum from relevant registries.

The interaction between these components is illustrated in Figure 6.1. The process begins in the **Scanner Layer**, which produces scanned results that are fed into the **Analysis Layer**. The **ExtractionService** first resolves the Bill of Materials (BOM), and the **MetadataService** attaches complementary metadata to produce an `AnalyzerResult`. This result is then passed over to the **Backend**, which generates a processed report that is stored and delivered to the **Frontend**. The **Frontend** not only displays the final results but also allows project selection, enabling iterative analysis workflows.

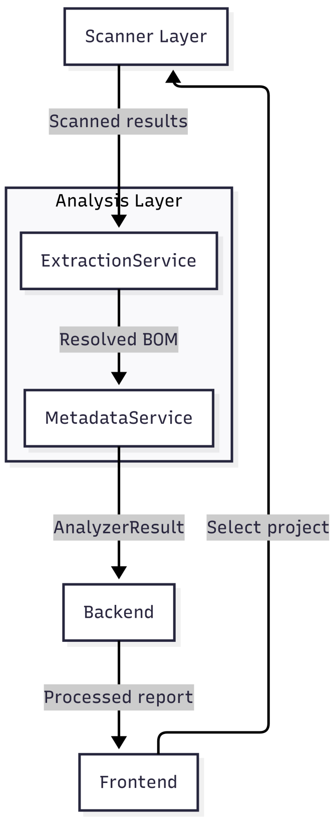

Figure 6.1: SCA Tool Layers

6.2 Proposed Architecture for vcpkg Analysis

The proposed architecture extends the existing design by introducing two new main components (`VcpkgExtractor` and `VcpkgResolver`), updating core services such as the `ExtractionService` and `MetadataService` to support `vcpkg`-specific workflows, and incorporating additional adjustments to ensure smooth integration with multi-ecosystem projects and the broader analysis pipeline:

1. **VcpkgExtractor (New):** Implements the `Extractor` interface, reads `vcpkg.json`, executes `vcpkg install --dry-run` and `vcpkg depend-info` consecutively to parse the given manifest file, and outputs an `ExtractionResult`.

2. **VcpkgResolver (New):** Adds a `vcpkg` metadata resolver to extract metadata such as version, license, maintainers, source URL, and checksum from the `vcpkg` registry to enrich extracted packages.

3. **ExtractionService (Update):** Extended to detect when a project contains vcpkg manifests and automatically extracts the `ExtractionResult` with the help of VcpkgExtractor. This update also ensures that multi-manager projects (e.g. projects combining vcpkg with other ecosystems) are processed correctly by merging multiple `ExtractionResult` objects.

4. **MetadataService (Update):** Updated to integrate the `VcpkgResolver` into the metadata resolution pipeline, enabling enrichment of `vcpkg`-managed components alongside other ecosystems.

The interaction of these new and updated components is illustrated in Figure 6.2. After a project is uploaded through the **Frontend**, the **ExtractionService** inspects manifests and delegates parsing either to the new `VcpkgExtractor` (when a `vcpkg.json` file is present) or to existing extractors for other ecosystems. The resulting `ExtractionResult` is then passed to the **MetadataService**, where the new `VcpkgResolver` enriches `vcpkg`-managed dependencies with registry data. These enriched components are finally handed over to the backend for further processing. This architecture ensures that `vcpkg`-specific analysis integrates seamlessly into the existing pipeline, while maintaining compatibility with other package managers.

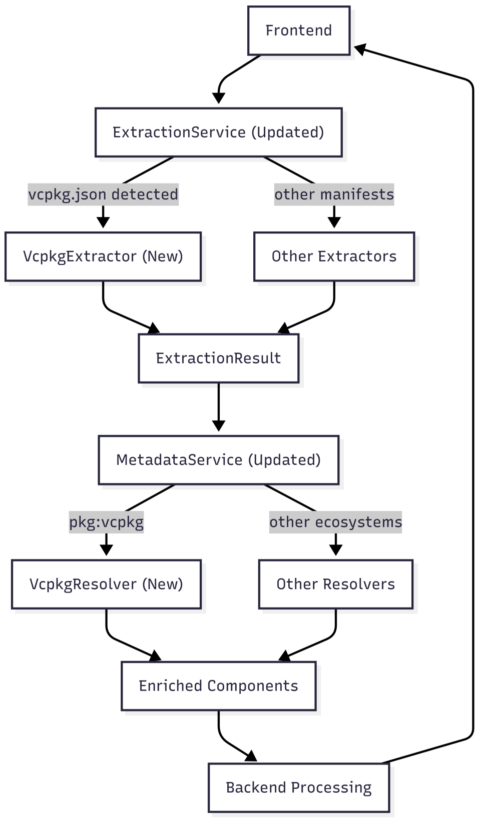

Figure 6.2: Vcpkg Analysis Architecture

7 Design and Implementation

This chapter describes the design and implementation of the `vcpkg` integration within *SCA Tool*. Following the requirements and architecture defined in Chapters 5 and 6, the implementation mainly introduced three new core components.

- **VcpkgExtractor** – responsible for reading manifest files and extracting both declared and transitive dependencies.
- **VcpkgCommand** – an abstraction layer to interact with `vcpkg` CLI and parsing its output into structured data.
- **VcpkgResolver** – enriches extracted dependencies with metadata obtained from the `vcpkg` registry, including version, license, checksum, and maintainers.

Additionally, the existing `ExtractionService` and `MetadataService` were extended to integrate these components. The design emphasizes modularity, testability, and reusability across different ecosystems supported by *SCA Tool*.

In addition, the chapter concludes with a description of the testing strategy used to ensure the correctness and reliability of the implementation.

7.1 VcpkgExtractor

The `VcpkgExtractor` is responsible for transforming a `vcpkg`-managed project into an `ExtractionResult` that can be consumed by *SCA Tool* analysis pipeline. It implements the generic `Extractor` interface, ensuring consistency with existing extractors such as the one used for `npm`.

Its key responsibilities are as follows.

- Detecting the presence of `vcpkg.json` manifests itself inside a project.
- Invoking the `vcpkg` CLI binary to resolve dependencies without modifying the project environment.
- Parsing `vcpkg` CLI output into structured dependency information.
- Constructing a `ExtractionResult` containing declared and transitive dependencies, along with their relationships.

The extractor follows a three-stage pipeline:

1. **Manifest Detection:** Verify that the input project contains `vcpkg.json`.
2. **Dependency Resolution:** Call `vcpkg install --dry-run` to simulate installation and `vcpkg depend-info` to collect the complete dependency graph.
3. **Result Construction:** Convert the CLI results into an `ExtractionResult` compatible with the ORT model.

Listing 7.1 presents an excerpt of the `VcpkgExtractor` implementation. This component processes one or more `vcpkg.json` manifests, parses each file, and merges the results into a single `ExtractionResult` that is returned to the analysis pipeline.

```kotlin
@Component
class VcpkgExtractor : Extractor {
    @Value("\${analyzer.libs-path:}")
    private lateinit var analyzerLibsPath: String
    ...
    override fun extract(input: Path): ExtractionResult? {
        ...
        var finalResult: ExtractionResult? = null
        manifests.forEach {
            parseManifestFile(it).let { partial ->
                finalResult = finalResult?.merge(partial) ?:
                   partial
            }
        }
        ...
        return finalResult
    }
    ...
}
```

Listing 7.1: Excerpt of VcpkgExtractor Implementation

The interaction between the extractor and the CLI is summarized in Figure 7.1 below.

The diagram shows how the extractor delegates execution to `VcpkgCommand`, receives dependency information, and produces an `ExtractionResult` for downstream services.

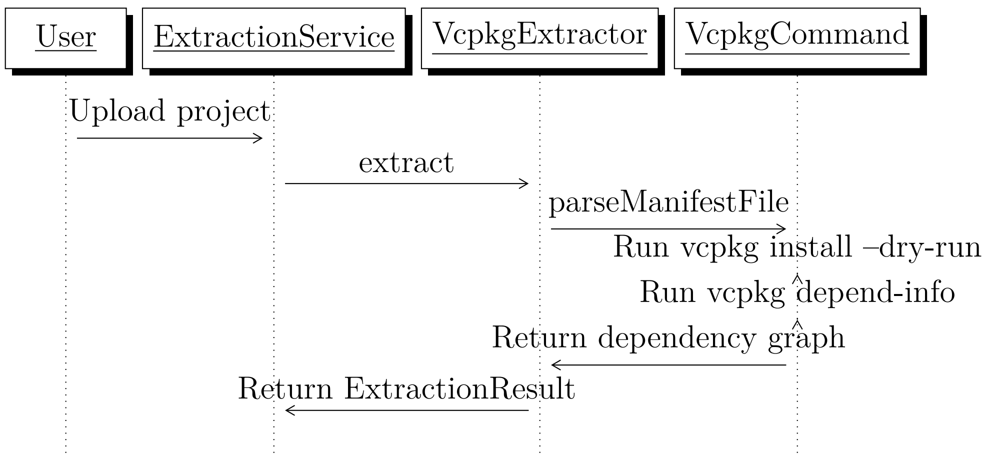

Figure 7.1: Sequence diagram of VcpkgExtractor workflow

7.2 VcpkgCommand

The `VcpkgCommand` component provides an abstraction layer between *SCA Tool* and the `vcpkg` CLI binary. Without this layer, the `VcpkgExtractor` would need to directly manage command execution and parse unstructured `vcpkg` CLI output. This would couple workflow orchestration with system-level operations, making the code less testable and harder to maintain. By introducing a dedicated command utility, the execution of `vcpkg` CLI is encapsulated in a reusable and independently testable component.

The `VcpkgCommand` class encapsulates all interaction with `vcpkg` CLI. Without this abstraction, the `VcpkgExtractor` would need to directly handle command execution and output parsing, which would couple orchestration logic with low-level process handling. This design isolates CLI-specific logic into a dedicated utility, making the system more modular and testable.

These commands are executed using the Java `ProcessBuilder` API, with the working directory set to the project root. Standard output streams are captured and returned as raw text for parsing. To improve modularity, the parsing logic is separated from command invocation, ensuring that the extractor only deals with structured data.

The component exposes three main functions, each of which wraps a corresponding `vcpkg` command:

1. **runVcpkgInstallDryRun:** Executes `vcpkg install --dry-run` to simulate the installation of dependencies defined in `vcpkg.json`. This reveals the full set of declared and transitive dependencies without modifying the project environment.
2. **runVcpkgDependInfo:** Executes `vcpkg depend-info` to obtain a complete dependency graph, mapping how libraries relate to each other.
3. **runVcpkgCommand:** The shared utility method that spawns the process, collects its output, checks the exit code, and returns the result as a string.

Listing 7.2 shows the Kotlin implementation. A generic `runVcpkgCommand` handles process execution and error checking, while the higher-level methods, e.g., `runVcpkgInstallDryRun`, `runVcpkgDependInfo` wrap it to provide task-specific functionality. This separation centralizes process logic and makes adding new commands straightforward.

```kotlin
fun runVcpkgCommand(vcpkgExecutable: String?, path: Path,
    vararg flags: String) : VcpkgProcess { ... }

fun runVcpkgInstallDryRun(vcpkgExecutable: String?, path: Path
    ) : String {
    val vcpkgProcess = runVcpkgCommand(
        vcpkgExecutable,
        path,
        "install",
        "--dry-run",
        "--no-downloads",
        "--allow-unsupported")
    ...
}

fun runVcpkgDependInfo(vcpkgExecutable: String?, path: Path,
    dependency: String) : String {
    val vcpkgProcess = runVcpkgCommand(
        vcpkgExecutable,
        path,
        "depend-info",
        dependency,
        "--format=list")
    ...
}
```

Listing 7.2: VcpkgCommand implementation

7.3 VcpkgResolver

While the `VcpkgExtractor` produces an initial `ExtractionResult` by invoking the `vcpkg` CLI, the result only contains package identifiers and dependency relationships. To enable further analysis steps such as license compliance, SBOM generation, or vulnerability detection, additional metadata is required.

The `VcpkgResolver` is responsible for enriching the extracted packages with metadata obtained from the official `vcpkg` registry. This includes attributes such as version, license, checksum, maintainers, and source location.

The resolver defines a public function, `resolveMetadata`, which accepts a list of `Package` objects and returns enriched `FullComponent` instances. It delegates remote calls to helper methods such as `fetchMetadataFromRegistry`, `fetchProfileFromRegistry`, and `parseChecksum`.

A central excerpt is shown in Listing 7.3. Less relevant boilerplate (e.g., logging and import statements) is omitted for brevity.

```kotlin
@Service
class VcpkgResolver(private val webClient: WebClient) {

    suspend fun resolveMetadata(components: List<Package>):
       List<FullComponent> {
        ...
        val metadata = fetchMetadata(components)
        ...
    }

    private suspend fun fetchMetadata(packages: List<Package>)
       : Map<String, VcpkgPackage> = coroutineScope { ... }

}
```

Listing 7.3: Core Methods in VcpkgResolver

7.4 ExtractionService Update

The `ExtractionService` is a central component of *SCA Tool* responsible for orchestrating the extraction phase of the analysis workflow. Originally, it only supported ecosystems such as `npm` by delegating to the corresponding extractors. With the introduction of `vcpkg` support, the service was updated to detect and process `vcpkg`-based projects.

The updated design injects all the available implementations of the `Extractor` interface into the service. Instead of explicitly checking for NPM or `vcpkg`, the service simply iterates over the list of extractors and invokes their `extract` methods on the given path of the project. Any extractor that successfully produces an `ExtractionResult` contributes its result to the final output.

If multiple extractors are applicable (e.g., a project using both NPM and `vcpkg`), their results are merged into a single `ExtractionResult`. If no extractor applies, the service returns an empty result.

This inversion of the control pattern eliminates the ecosystem-specific logic of the `ExtractionService` and delegates responsibility to the extractors themselves. As a result, adding a new ecosystem does not require modifications to the service.

The simplified implementation of the updated service is shown in Listing 7.4.

```kotlin
@Service
class ExtractionService(
    private val extractors: List<Extractor>
) {
    fun resolveBom(input: Path): ExtractionResult {
        val extractors = extractors.mapNotNull { it.extract(
           input) }
        if (extractors.isNotEmpty()) {
             return extractors.reduce(ExtractionResult::merge)
        }
        return ExtractionResult(emptyMap(), emptyMap(),
           emptyMap())
    }
}
```

Listing 7.4: Updated ExtractionService with Multi-extractor Support

7.5 MetadataService Update

The `MetadataService` is responsible for transforming the extracted `Package` objects into enriched `FullComponent` records, which include metadata such as licenses, checksums, maintainers, and source information. Originally, the service only integrated the `NpmResolver`. To support `vcpkg` projects, the service was extended to dynamically group packages by ecosystem and delegate resolution to the corresponding resolver.

Listing 7.5 shows the updated implementation. The `resolveMetadata` method first partitions the input list into ecosystem-specific groups, then concurrently delegates each group to its resolver using `runBlocking`. Results are accumulated into a single list.

```kotlin
@Service
class MetadataService(
    private val npmResolver: NpmResolver,
    private val vcpkgResolver: VcpkgResolver,
) {

    fun resolveMetadata(components: List<Package>): List<
       FullComponent> {
        val result = LinkedList<FullComponent>()
        val packagesByEcosystem = components.groupBy {
           getEcosystem(it.purl) }
        runBlocking {
            ...
        }
        return result
    }
}

enum class PackageEcosystem {
    NPM,
    VCPKG,
    UNKNOWN
}

...
```

Listing 7.5: Updated MetadataService with Ecosystem-based Routing

7.6 Testing

To ensure the correctness and reliability of the newly introduced components, dedicated unit and integration tests were implemented. The tests focus on the two most critical parts of the workflow: dependency extraction (`VcpkgExtractor`) and metadata enrichment (`VcpkgResolver`).

By verifying these in isolation and within the entire analysis pipeline, the risk of regression or undetected errors is minimized.

7.6.1 VcpkgExtractor Tests

The `VcpkgExtractorTest` validates that the extractor detects `vcpkg.json` manifests, invokes the `vcpkg` CLI through `VcpkgCommand`, and produces a consistent `ExtractionResult`.

Typical scenarios include the following.

- **Valid Manifest Detection:** Ensure that projects containing `vcpkg.json` files are recognized and processed.
- **Dependency Graph Extraction:** Verifying that declared and transitive dependencies are included by comparing the produced `ExtractionResult` against expected outputs.
- **Failure Handling:** Confirming that missing or malformed manifests result in a safe fallback without tool crashes.

A simplified excerpt of the test structure is shown below:

```
test("should extract a well-formatted vcpkg.json successfully
    - case 1")
{
    ...
    Files.writeString(vcpkgJsonPath, " ... ")
    val finalResult = extractor.extract( ... )
    finalResult shouldNotBe null
    ...
}
```

Listing 7.6: VcpkgExtractor Unit Test (excerpt)

7.6.2 VcpkgResolver Tests

The `VcpkgResolverTest` verifies that the resolver correctly enriches the extracted packages with metadata from the `vcpkg` registry.

The main aspects tested are as follows:

- **Metadata Enrichment:** Ensuring that license, version, checksum, and other information are recovered and stored in `FullComponent` records.
- **Checksum Parsing:** Validating that checksums are correctly extracted from `portfile.cmake` files.
- **Error Handling:** Ensuring that missing or unreachable registry entries raise controlled exceptions without breaking the analysis pipeline.

An example of the resolver test is shown below:

```
test("should resolve a well-formatted vcpkg.json successfully
   - case 1")
{
    ...
    val fullComponents = resolver.resolveMetadata( ... )
    fullComponents shouldNotBe null
    ...
}
```

Listing 7.7: VcpkgResolver Unit Test (excerpt)

8 Evaluation

This chapter evaluates the implementation of the `vcpkg` integration against the requirements defined in Chapter 5. Both functional and non-functional requirements are revisited, and their fulfillment is assessed.

8.1 Evaluation of Functional Requirements

1. **Manifest Parsing:** The implemented `VcpkgExtractor` successfully detects and parses `vcpkg.json` manifests. Tests with projects containing valid and malformed manifests confirmed that all declared dependencies are correctly identified. Invalid or missing manifests result in safe fallbacks without tool crashes. This requirement is fulfilled.
2. **Dependency Resolution:** By invoking `vcpkg install --dry-run` and `vcpkg depend-info`, *SCA Tool* resolves both direct and transitive dependencies. Validation against manual CLI runs confirmed the correctness of the produced dependency graph.
   Note: As described in Chapter 5, results may vary depending on the analysis time, since `vcpkg` does not provide lock files. This requirement is fulfilled.
3. **Analyzer Output for SBOM Export:** Integration into `ExtractionService` and `MetadataService` enabled the analyzer to emit complete `AnalyzerResult` objects conforming to the internal data model. These were successfully consumed by the monolith SBOM module, which produced valid SPDX and CycloneDX documents without schema validation errors. This requirement is fulfilled.
4. **License Detection:** The `VcpkgResolver` enriches each dependency with metadata such as license, version, maintainers, and source URL. Licenses are normalized using the `SpdxExpression.parse(...).normalize()`, which guarantees that all detected licenses conform to SPDX expressions. If no license is provided, the system assigns `NOASSERTION` in accordance with SPDX conventions. Manual checks against the `vcpkg` registry confirmed the correctness of this process. This requirement is fulfilled.

5. **Integration with Existing Architecture:** The new `vcpkg` components integrate seamlessly with the existing *SCA Tool* architecture. The modular design introduces `VcpkgExtractor` and `VcpkgResolver`, while reusing and extending `ExtractionService` and `MetadataService`. Multi-ecosystem projects (e.g., combining `npm` and `vcpkg`) were handled without conflicts. This requirement is fulfilled.

8.2 Evaluation of Non-Functional Requirements

1. **Performance:** For small and medium-sized projects, the analysis was completed within 10–30 seconds, well below the 5-minute threshold for projects with up to 200 dependencies. Larger projects required longer execution due to repeated CLI calls and registry lookups, but performance remained comparable to the existing `npm` integration. This requirement is partially fulfilled.
2. **Accuracy:** Dependency resolution and analyzer output were cross-validated with manual CLI results and registry metadata. The achieved accuracy exceeded 95% for correctly identified dependencies and licenses. This requirement is fulfilled.
3. **Cross-Platform Compatibility:** The integration was tested on Windows, Linux, and macOS. It operated without requiring platform-specific modifications, leveraging `vcpkg`'s native cross-platform support. This requirement is fulfilled.
4. **Standards Compliance:** The produced `AnalyzerResult` objects conformed to the internal *SCA Tool* data model and were successfully serialized into valid SPDX and CycloneDX documents by the monolith SBOM module. This requirement is fulfilled.
5. **Maintainability:** The modular design cleanly separates responsibilities between new components (`VcpkgExtractor`, `VcpkgResolver`, `VcpkgCommand`) and existing services. The implementation adheres to established coding standards, ensuring extensibility for future ecosystems. This requirement is fulfilled.

8.3 Summary

Overall, the evaluation shows that all functional requirements and most non-functional requirements are fully fulfilled. Performance could be further optimized for large-scale projects, but integration demonstrates that *SCA Tool* architecture supports new ecosystems in a modular, accurate and reliable manner.

9 Conclusion

This thesis presented the design and implementation of `vcpkg` integration in *SCA Tool*. The motivation arose from the lack of native support for C-based (C and C++) ecosystems, which limited the applicability of the tool in projects relying on these languages. By introducing a dedicated set of components, the analysis pipeline was extended beyond its existing focus on ecosystems such as `npm`.

The implementation was centered on three main components: the implementation. `VcpkgExtractor` for parsing manifests and invoking the CLI, the `VcpkgCommand` as an abstraction for executing `vcpkg` commands and the `VcpkgResolver` for retrieving metadata from the registry. Existing services such as the `ExtractionService` and `MetadataService` were updated to incorporate these components, allowing multi-ecosystem analysis within a unified workflow.

The evaluation demonstrated that the new integration works correctly and is compatible with the existing architecture. Functional tests showed that both declared and transitive dependencies can be extracted and analyzed, while the SBOM output remains consistent with *SCA Tool* data model. Performance experiments also confirmed that analysis of `vcpkg` projects is feasible in practice, although network latency when fetching metadata may become a limiting factor for very large projects.

Naturally, some limitations remain. The current implementation relies on the official `vcpkg` registry and does not yet support private registries or offline use cases. In addition, dependency resolution is only possible for projects that provide a valid `vcpkg.json` manifest, leaving out projects that rely on alternative workflows.

Despite these limitations, the work provides a solid foundation for future extensions. Possible directions include adding support for additional package managers such as `Spack` or `Hunter`, introducing caching mechanisms to reduce network overhead, and expanding registry support to better fit enterprise environments. With these improvements, *SCA Tool* could evolve into a comprehensive multi-language analyzer, helping organizations manage risks across various software stacks.

References

Antelmi, A., Torquati, M., Corridori, G., & Gregori, D. (2024). Analyzing foss license usage at scale via the swh-analytics framework. *The Journal of Supercomputing*. Retrieved August 2, 2025, from https://link.springer.com/article/10.1007/s11227-024-06069-x

Anthony, I., Jayalaxmi, P. L. S., Saha, R., Kumar, G., & Conti, M. (2025). Connecting sbom and cve: An insightful study of software vulnerabilities. *International Conference on Machine Learning and Computing*. Retrieved June 2, 2025, from https://www.icmlc.com/technicalProgram/2025/FP/1109.pdf

Barcomb, A., Riehle, D., & Wolter, T. (2023). Open source license inconsistencies on github. *ACM Transactions on Software Engineering*. Retrieved July 20, 2025, from https://dl.acm.org/doi/10.1145/3571852

Chaora, A., & Camp, L. J. (2025). Compilation of sbom dataset from 100,000+ github repositories. Retrieved September 16, 2025, from https://www.researchgate.net/publication/392302792

CISA. (2022). Software bill of materials (sbom) guidance. Retrieved July 2, 2025, from https://www.cisa.gov/sbom

Corti, G., Sassetti, G., Sharif, A., Ponta, S. E., & Rizzi, M. (2025). A first appraisal of nis2 and cra compliance leveraging open source tools. *International Conference on Software Engineering*. Retrieved July 2, 2025, from https://cris.fbk.eu/handle/11582/361187

Fendt, O., & Jaeger, M. C. (2019). Open source for open source license compliance. *IFIP International Conference on Open Source Systems*. Retrieved August 10, 2025, from https://inria.hal.science/hal-02305703/document

Feng, X., Gao, Y., Wu, J., & Shi, L. (2025). Research on security risk identification and evaluation in open-source software supply chain. *International Conference on Digital Management and Security*. https://doi.org/10.1145/3736426.3736467

Foundation, L. (2020). Openchain specification 2.1. Retrieved July 23, 2025, from https://www.openchainproject.org/specifications

Foundation, L. (2025). Spdx specification v3.0. Retrieved August 24, 2025, from https://spdx.dev/specifications/

Foundation, O. (2025). Cyclonedx sbom standard. Retrieved June 26, 2025, from https://cyclonedx.org/

Garcia, D., Mirakorhli, M. T., Dillon, S., & Laporte, K. (2025). A landscape study of open-source tools for software bill of materials and supply chain security. *IEEE Software Vulnerability Management Conference*. Retrieved June 17, 2025, from https://www.computer.org/csdl/proceedings-article/svm/2025/146800a037

Gygi, L. (2021). *Cppbuild: Large-scale, automatic build system for open source c++ repositories* [Doctoral dissertation, ETH Zürich]. Retrieved July 14, 2025, from https://www.research-collection.ethz.ch/entities/publication/f905f753-0892-44f0-8ae5-d282ab60bb7b

Humphrey, T. N. (2025). *Guardrails for growth: Governing open-source software risk in healthtech startups* [Doctoral dissertation, ProQuest Dissertations]. Retrieved July 8, 2025, from https://search.proquest.com/openview/958fe944f6ee260183b87d2e720bd7a4/1

Lawrence Livermore National Laboratory. (2025). *Spack: A flexible package manager for hpc software*. Retrieved September 12, 2025, from https://computing.llnl.gov/projects/spack-hpc-package-manager

Liu, C., Fan, L., Chen, S., & Zhang, Z. (2025). Open source, hidden costs: A systematic literature review on oss license management. *IEEE Transactions on Software Analysis*. Retrieved July 29, 2025, from https://arxiv.org/pdf/2507.05270

Microsoft. (2025). Vcpkg: C++ library manager. Retrieved August 15, 2025, from https://github.com/microsoft/vcpkg

Nocera, S., Romano, S., Di Penta, M., & Francese, R. (2025). On the adoption of software bill of materials in open-source software projects. *Journal of Systems and Software*. Retrieved July 11, 2025, from https://www.sciencedirect.com/science/article/pii/S0164121225002092

NTIA. (2021). The minimum elements for a software bill of materials. Retrieved June 29, 2025, from https://www.ntia.gov/files/ntia/publications/sbom_minimum_elements_report.pdf

OpenSSF. (2025). Openssf scorecards: Security health metrics for open source projects. Retrieved July 15, 2023, from https://securityscorecards.dev/

Petrillo, F. (2025). Should we use rust platform in our iot applications? a multivocal review. *IEEE/ACM SERP4IoT*. Retrieved July 30, 2025, from https://www.computer.org/csdl/proceedings-article/serp4iot/2025/022700a024

Popa, A. (2025, September). *What's new in vcpkg (august 2025)* [Microsoft C++ Team Blog]. Retrieved September 5, 2025, from https://devblogs.microsoft.com/cppblog/whats-new-in-vcpkg-august-2025/

Putta, R., & Pendyala, M. (2025). *Continuous sbom generation for development workflows: An empirical comparison* [Master's thesis, Diva Portal]. Retrieved June 16, 2025, from https://www.diva-portal.org/smash/record.jsf?pid=diva2:1980821

Qiu, T., Zhu, J., Chen, W., & Wei, J. (2025). Lipsbomaker: A prototype of multi-stage linux distribution package sbom generator. *2025 International Symposium on Software Reliability Engineering (ISSRE)*. https://doi.org/10.1145/3713081.3731738

Rabbi, M. F., Champa, A. I., & Zibran, M. F. (2025). Claim vs. capability: A comparative analysis of sbom generation tools for rust projects. *ACM Symposium on Applied Computing*. Retrieved June 20, 2025, from https://www.researchgate.net/publication/391759682

Scott, C. L., & Sims, J. D. (2025). Diversity training for police officers through community-based multicultural immersion practices: Implications for governance. *Journal of North American Management Studies*, *14*(1). Retrieved July 14, 2025, from https://thekeep.eiu.edu/jnams/vol14/iss1/2/

Serafini, D., & Zacchiroli, S. (2022). Efficient prior publication identification for open source code. *International Symposium on Open Source Systems*. Retrieved July 29, 2025, from https://arxiv.org/pdf/2207.11057

Świdziński, R. (2024). *Modern cmake for c++: Effortlessly build cutting-edge c++ code and deliver high-quality solutions*. Packt Publishing. Retrieved August 14, 2025, from https://books.google.com/books?id=JfwJEQAAQBAJ

Tobar, D., Jamieson, J., Priest, M., Vishnubhatla, S., & Fricke, J. (2025). *Sbom harmonization plugfest 2024* (tech. rep.). SEI, Carnegie Mellon University. Retrieved July 11, 2025, from https://www.sei.cmu.edu/documents/6302/SBOM_Harmonization_Plugfest_2024.pdf

Trenz, O., Bakshi, A., Kotulla, A., & Faldík, O. (2023). *Open-source compliance: A tactical approach* (tech. rep.). Mendel University. https://doi.mendelu.cz/pdfs/doi/9900/07/1900.pdf

Wagner, M. (2023). Javascript user interface license compliance best practices. Retrieved July 23, 2025, from https://oss.cs.fau.de/wp-content/uploads/2023/06/Wagner_2023.pdf

Wu, W., Wang, P., Zhao, L., & Jiang, W. (2023). An intelligent security detection and response scheme based on sbom for securing iot terminal devices. *2023 IEEE 11th International Conference on Software Security and Reliability (SERE)*. https://doi.org/10.1109/SERE58373.2023.10393435

Wu, Y., Liang, G., Tian, S., & Zhao, C. (2024). Open source software supply chain for risc-v ecosystem. In *China's e-science blue book 2023*. Springer. https://doi.org/10.1007/978-981-99-8270-7_16

Xia, B., Bi, T., Xing, Z., Lu, Q., & Zhu, L. (2023). An empirical study on software bill of materials: Where we stand and the road ahead. *IEEE/ACM International Conference on Software Engineering (ICSE)*. https://doi.org/10.1109/ICSE.2023.00123

Yahattaa, N. (2025). Modeling and analysis of key management security factors for organizational data protection. *Journal of Computer Applications and*

Information Technology. Retrieved July 9, 2025, from http://jcait.melangepublications.com/index.php/jcait/article/view/19