ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

DEPARTMENT OF COMPUTER SCIENCE
AND ENGINEERING

ARTIFICIAL INTELLIGENCE

MASTER THESIS

in

Fundamentals of Artificial Intelligence and Knowledge Representation

A SEMANTIC FRAMEWORK FOR MODELING AND ANALYSING SOFTWARE SUPPLY CHAINS THROUGH SOFTWARE BILL OF MATERIALS

CANDIDATE
Gianluca De Bonis

SUPERVISOR
Prof. Federico Chesani

CO-SUPERVISOR
Giacomo Tenaglia (CERN)

Academic year 2024-2025

Session 3rd

dedicated(X) :- friend(X).

Abstract

This thesis offers a multi-layered semantic structure for modeling and examining software supply chains using Software Bills of Materials (SBOMs). Contemporary software ecosystems depend significantly on open-source components, but SBOMs only offer standalone snapshots of elements, missing integrated perspectives on organisational context, vulnerability propagation, and internal software behaviour. This study integrates Semantic Web technologies, graph-based dependency modeling, and function-level structural analysis to overcome these limitations. At the organisational level, diverse SBOMs, survey data, licensing details, and vulnerability records are integrated into an ontology-based knowledge graph, facilitating expressive queries and automated reasoning throughout varied software landscapes. At the project level, the Vulnerability-Dependency Graph (VDGraph) model integrates SBOM dependency details with vulnerability information from Software Composition Analysis (SCA) tools, aiding the analysis of how vulnerabilities spread through dependency chains. Ultimately, at the code level, function-call graphs described by node centrality metrics and Graph Attention Network (GAT) embeddings reflect the structural significance of functions within an application, providing insights on how updates in dependencies might influence internal behavior. Created during an internship at CERN's Open Source Program Office, this framework offers a complete, scalable method for understanding, managing, and safeguarding intricate software supply chains within large and heterogeneous organisations.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Nowadays, software development relies heavily on large ecosystems of open-source libraries, making the software supply chain more and more complex. Even though this model reliant on dependencies speeds up development, it also brings notable difficulties: undiscovered vulnerabilities in transitive dependencies, inconsistent licensing data, and limited insight into how any software components can potentially spread risk throughout an organisation. To tackle these challenges, in recent years Software Bills of Materials (SBOMs) have emerged as an essential tool for enhancing transparency and facilitating automated data processing regarding software composition [28]. However, in reality, SBOMs by themselves fall short: they offer detached views of dependencies but lack support for semantic integration, cross-project assessments, or studies on vulnerability propagation.

This thesis proposes a semantic framework for modelling and analysing software supply chains using SBOMs, combining knowledge engineering, graph-based dependency modelling, and function-level structural analysis. The objective is to offer a unified perspective that spans from organisational insights across many projects down to fine-grained reasoning about the internal structure of a single software application.

As a first level of abstraction, ontologies and Semantic Web technologies

[7, 1] are employed to merge diverse SBOM standards, organisational metadata, license data, and vulnerability records into a unified knowledge graph. This offers a wide perspective of the software environment: identifying the most utilized technologies, how dependencies extend throughout organisational divisions, how licenses influence one another, and which software services could be impacted by known vulnerabilities. Ontologies enable us to merge various SBOM standards alongside organisational data, enriching them with domain knowledge pertinent to software governance and security. With this semantic layer SBOMs can become, from being isolated artifacts, interlinked elements of a broader knowledge framework that supports expressive queries and automated reasoning.

From this global perspective, the second section of the study focuses on more detailed analysis of specific projects. To achieve this, this project utilizes the VDGraph model [40], a graph representation that combines SBOM dependency data with vulnerability information gathered from Software Composition Analysis (SCA) tools. While the ontology-driven knowledge graph enables reasoning across various projects, VDGraph "zooms in" on a specific code base (or a portion of it) and thoroughly illustrates how vulnerabilities spread through dependency chains, a topic often studied in existing research on dependency-level vulnerability propagation [22]. This helps experts address questions like: which dependencies connect a project to a vulnerable component? Are vulnerabilities more often present in direct or transitive dependencies? Which libraries can be identified as essential propagation points for vulnerabilities in the dependency structure? All these questions can be addressed thanks to VDGraph and the framework later described.

The final level of granularity is provided by a study based on Graph Attention Networks (GATs) [38] applied to function-level call graphs, again of a single project. Although VDGraph helps in understanding the structure and role of external dependencies, it does not clarify how updates or vulnerabilities could affect the internal functioning of an application. To investigate

this issue, this project represents the program's internal architecture as a directed function-call graph and derives structural characteristics to define each function. Employing a GAT facilitates a learned understanding of structural significance: rather than considering all neighbouring functions as equally essential, the attention mechanism determines which connections are most impactful during information aggregation. The generated embeddings assist in finding architectural entry points, functionally central components, and areas of the codebase that might be more susceptible to dependency modifications [39]. This is the deepest degree of examination within the framework: extending from ecosystem-level reasoning (ontologies), to project-level dependency analysis (VDGraph), and ultimately to internal function-level structure (GAT).

The context in which this thesis was developed is during an internship at the Open Source Program Office (OSPO) of CERN, the European Organization for Nuclear Research, where there is wide usage and development of open-source software. The objective of this project is to help service managers, OSPO and Computer Security Team members in their workflows.

Essentially, this work integrates the three analytical levels mentioned above into a unified framework that facilitates both organisational supervision and project-level analysis. Throughout this project, I created a modular set of ontologies to semantically merge SBOM data, organisational metadata, licensing details, and vulnerability records into a unified knowledge graph; I developed a complete data processing pipeline that transforms survey responses and SBOMs into RDF, links them to external resources, and allows for expressive SPARQL querying across CERN's software ecosystem; I utilized the VDGraph model on specific projects to investigate how vulnerabilities spread through dependency chains; finally, I deepened the analysis to the internal structure of applications by constructing function-level call graphs described

with node centrality metrics and GAT embeddings to capture structural importance within the codebase. Collectively, these contributions create a comprehensive semantic structure that provides extensive visibility and detailed understanding of software supply chains, aiding OSPO members, service managers, and computer security teams in overseeing and managing the software landscape at CERN.

The structure of the thesis follows a logical progression from conceptual foundations to practical implementation. Chapter 2 offers the necessary background to understand the work, covering ontologies, Semantic Web technologies, knowledge graphs, SBOM standards, VDGraph, and GATs. Chapter 3 details the methodology, addressing ontology creation, dataset development, the data processing workflow, project-level graph modeling, and the function-level analysis tool; it also presents some findings achieved during these phases. Chapter 4 provides concluding thoughts, encapsulating the work's contributions and exploring potential directions for enhancements in software supply chain analysis and vulnerability management.

Chapter 2

Background

2.1 CERN

The European Organization for Nuclear Research, commonly referred to by its French acronym **CERN (Conseil Européen pour la Recherche Nucléaire)**, is one of largest and most respected centres for scientific research in the world. Founded in 1954 by twelve European countries, its primary mandate is to design, build, and operate the complex systems required for carrying out high-energy particle physics experiments, while also providing an open environment for research and innovation in computing, engineering, and data science.

In particular, CERN is home to the **Large Hadron Collider (LHC)**, the world's largest and most powerful particle accelerator, and a complex infrastructure for storage and computations that is needed for managing the immense amount of data created by its experiments. Each year, the LHC generates several tens of petabytes of raw data, that need to be filtered, stored, and analysed, all by a wide community of researchers distributed globally. To meet these unique computational demands, CERN operates one of the most advanced data centres in the world and coordinates the Worldwide LHC Computing Grid (WLCG), a distributed infrastructure connecting more than 170 computing centres across 40 countries. This system provides the collective processing and storage power necessary to transform experimental data into

scientific knowledge.

Beyond fundamental physics, CERN has had a deep impact on technology itself and society. In particular, Tim Berners-Lee proposed and implemented the **World Wide Web (WWW)** in 1989 while working at CERN, with the intention of easily enabling information sharing among scientists. This invention transformed international communication and continues to serve as a prime illustration of the revolutionary technological advancements that may be produced through open scientific collaboration.

Some of the technologies he proposed, like the Semantic Web [7], are at the core of this thesis.

2.1.1 Open Source Program Office (OSPO)

Since its inception, knowledge sharing has been at the core of CERN, with the pure spirit of letting knowledge grow when let free among the community. The CERN Convention itself, signed on July 1st 1953, stated the importance of the principle of publishing and disseminating the results of its experimental and theoretical work in order to advance global scientific knowledge. And so it did. In the 1970s, CERN established a school of computing — still ongoing — to fill the competence gap between physicists and computer scientists. It shared HBOOK–a pioneering software for two-dimensional histogram analysis– with different other institutions, from the United States to Russia. It also released the capacitive touch screen, the technology at the base of modern smartphones, and, as previously mentioned, the World Wide Web.

In recent years CERN has increased its efforts in openness and collaboration by identifying the need of having a formal entity dedicated to advocate for it. After several studies on the topic, for example on vendor lock-in and licensing models, the **Open Source Program Office (OSPO)** has been established in 2023. It now serves as an interdepartmental body that is tasked with the implementation of the Open Science Policy regarding Open Source

Software and Hardware, promotes the organisation as a contributor to Open Source development and enables due diligence related to Open Source. In particular, part of the OSPO mandate includes:

- Provide recommendations for open-sourcing software and hardware;
- Consult, advise and train CERN's community on best practices, tools, licences, developments, etc. for open-source projects;
- Facilitate collaboration with external open-source communities and other research institutions.
- Provide a public catalogue of CERN's open-source software and hardware.

It's in this context that this work has been carried on, especially in trying to facilitate the work of the OSPO in their workflows of recommendations for open-sourcing and having an overview of the software developed within the organisation. Besides this, it has been considered useful by the Computer Security Team to take advantage of this effort to also overview and manage vulnerabilities and security issues across CERN.

2.1.2 CERN structure

Operationally, CERN is organised into a hierarchical and functional structure that ensures both clarity of responsibility and flexibility in project management:

- The Organization is divided into several **Departments**, each responsible for a specific domain such as accelerators, engineering, finance, human resources, research, and information technology.
- Within each Department, **Groups** handle specialised areas of activity. For example, the Information Technology (IT) department includes groups

responsible for computing devices, storage and data management, platforms and workflows, and so on.

- Each Group consists of one or more **Sections**, smaller units focusing on well-defined technical or operational domains. Sections are the fundamental organisational layer for managing services, technical systems, and personnel.

This structure provides a clear framework for accountability, while allowing for cross-departmental collaboration in large-scale projects such as LHC upgrades, detector construction, or IT infrastructure modernisation.

To manage the vast array of digital and operational services required to support its activities, CERN uses a commercial customer service platform as its central system for service management that follows the *ITIL (Information Technology Infrastructure Library)* framework [2, 11]. Within this framework, services are categorised in a hierarchical model designed to represent the relationships between business processes and the underlying technical components:

- **Service Areas** define broad operational domains (for example, IT, Accelerator, Finance, or Learning);
- Each Service Area is subdivided into **Customer Services**, which correspond to coherent service categories as perceived by the end users;
- **Service Elements** represent the actual end-user services — for instance, email delivery, software repository access, or network connectivity. Every Service Element is owned and maintained by a designated Section, ensuring accountability and clear responsibility boundaries.
- Each Service Element depends on one or more **Functional Elements**, which describe the underlying technical components or processes required to deliver the service.

2.2 Software Bill Of Materials

Software systems are typically assembled from numerous external components—ranging from mature open-source libraries to small, specialised utilities. This practical approach, though, also introduces dangerous risks as the ones already mentioned of hidden vulnerabilities naturally embedded in libraries and license terms that bring legal and compliance risk to organisations.

For these reasons, **Software Bill of Materials (SBOM)** are now a fundamental building block to address these issues. An SBOM is a machine-readable list of all the constituent parts, libraries, and dependencies used in a piece of software. Like an ingredient label on food, an SBOM gives visibility into what's included in a software package, enabling better risk analysis, security practices, and compliance controls. From a vulnerability management perspective, SBOMs allow organisations to quickly map known vulnerabilities (such as those published in the NVD [23] or OSV [29] databases) to the specific versions of the components they are using [28]. This improves the time to respond to a threat and assists with prioritization of remediation based on the components most impactful to the business.

Besides vulnerabilities, SBOMs also enable one to list the open-source licenses used in each component. This facilitates due diligence, prevents conflicts of licenses, and maintains compliance with corporate or regulatory standards.

The most widely adopted SBOM formats are SPDX [35], CycloneDX [32], and SWID [18]. They all share the common goal of increasing transparency and traceability within the software supply chain, even if each of them was created to cover different use cases.

SWID (Software Identification Tag), standardized as ISO/IEC 19770-2, is a precursor to the modern concept of SBOMs and initially targeted asset and inventory management. SWID tags are XML documents that can be embedded directly into software packages or distributed with them. Even if not as

expressive as SPDX or CycloneDX in representing complex dependency relationships, SWID is still important for software identification and compliance reporting.

SPDX (Software Package Data Exchange) [34] is a stardard developed by the Linux Foundation and officially recognized as an ISO/IEC standard (ISO/IEC 5962:2021). It is the first general-purpose SBOM standard to be created, having much more expressivity than SWID which was the only way of identifying software at the time. Its main use case has always been the one of standardizing and simplifying license compliance workflows by providing a detailed way of describing software packages, their relationships, their licenses and all its related metadata. To help their purpose, part of the standard is the SPDX-License-Identifier, a unique way of identifying licenses in the code which is now adopted by CycloneDX as well. SPDX supports a wide variety of formats such as tag-value, JSON, YAML and in particular XML/RDF thanks to the fact that the standard is also described by an official OWL ontology [35].

The **CycloneDX** [32] standard, designed and maintained by the OWASP Foundation [31], has its main focus instead on software security. In particular it helps in vulnerability management, component analysis and software integrity verification. It is probably the preferred standard nowadays since security issues are the main concern during the software lifecycle, it is more simple and more lightweight. It supports different formats as JSON, XML and Protocol Buffers and there is a large ecosystem of tools for generation and analysis that support it, which often makes it the preferred choice for software developers.

In practice, SPDX and CycloneDX emerged as the two dominant standards for generating and exchanging SBOMs. Both are actively maintained, widely supported by open-source tools, and have backing from a number of government and industry initiatives — including some by the U.S. National

Telecommunications and Information Administration (NTIA) and the Cybersecurity and Infrastructure Security Agency (CISA). Choosing between them depends solely on the intended use cases: SPDX mainly for licensing and provenance tracking (that can be more in line with OSPOs use cases), while CycloneDX for risk handling and vulnerabilities management (more in line with Computer Security Team use cases).

To illustrate the structure and content of an SBOM, Listing 2.1 shows a simplified example in JSON CycloneDX containing metadata, components declaration and dependency representation.

```
{
  "bomFormat": "CycloneDX",
  "specVersion": "1.4",
  "version": 1,
  "metadata": {
    "component": {
      "type": "application",
      "name": "ExampleApp",
      "version": "1.0.0"
    }
  },
  "components": [
    {
      "type": "library",
      "name": "flask",
      "version": "2.3.2",
      "purl": "pkg:pypi/flask@2.3.2",
      "hashes": [
        {
          "alg": "SHA-256",
          "content": "2f3c7be8032..."
```

```
        }
      ]
    },
    {
      "type": "library",
      "name": "jinja2",
      "version": "3.1.2",
      "purl": "pkg:pypi/jinja2@3.1.2"
    },
    ...
  ],
  "dependencies": [
    {
      "ref": "pkg:pypi/flask@2.3.2",
      "dependsOn": [
              "pkg:pypi/jinja2@3.1.2",
              "pkg:pypi/werkzeug@2.3.7",
              "pkg:pypi/itsdangerous@2.1.2",
              "pkg:pypi/click@8.1.7",
              ...
      ]
    }
  ]
}
```

Listing 2.1: Example fragment of a simplified CycloneDX SBOM in JSON format

2.3 Knowledge Engineering

Knowledge engineering (KE) is the field of study that tackles the acquisition, modelling and organization of knowledge so that it can be processed by

computational systems. It emerged as a branch of Artificial Intelligence in response to the need of making implicit human knowledge explicit for computers. The main focus of the field is to enable systems to represent, reason about and exchange information in a consistent and explainable manner [37].

In practice, KE consists of identifying the most important elements of the domain under study, defining their relationships, rules and constraints that regulate their interactions. This is the starting point for later developing expert systems, intelligent agents and knowledge graphs.

In the context of software ecosystems, KE provides the means to structure information about software components, their dependencies, vulnerabilities, and lifecycle metadata in a uniform and interoperable way. More specifically, it is crucial when dealing with SBOMs and software supply-chain challenges: the landscape is extremely heterogeneous, with multiple standards, formats and tools. Ultimately, KE offers the methodology needed to unify these diverse sources into a coherent and semantically consistent model, enabling integration, reasoning, and the automation of security and compliance analysis.

2.3.1 Semantic Web

Originating from Knowledge Engineering, the Semantic Web extends its ideas of formal representation and reasoning into the open context of the World Wide Web. It was conceived by Tim Berners-Lee as a "web of data", where all the information is represented through formal semantics allowing autonomous agents to understand and reason over distributed knowledge in the web [6].

The key technologies involved in the Semantic Web are standardized by the World Wide Web Consortium (W3C), and they are:

- **RDF (Resource Description Framework)**: technology used to define directed graphs composed of triple statements in the form of $subject - predicate - > object$ where the subject is a node, the predicate is an edge and the object is either another node or a literal [4].

- **RDFS (RDF Schema)**: adds to RDF basic vocabulary for describing classes and properties, introducing concepts like hierarchies and simple constraints.
- **SPARQL**: query language used for querying data from RDF knowledge bases.
- **OWL (Web Ontology Language)**: builds upon the previously mentioned technologies to enable more complex logical definitions and reasoning.

This wide ecosystem allows data linkage and interoperability across heterogeneous domains, which is what we need in the context of integrating data with external sources such as vulnerability knowledge bases (OSV, NVD) and open-source project metadata. With the usage of these standards, SBOM-related information can be represented as part of a global graph of interlinked knowledge making it possible to trace dependencies, vulnerabilities and perform semantic queries that could go beyond the capabilities of relational databases.

2.3.2 Ontologies

In the field of knowledge engineering, an ontology is defined as a formal, explicit specification of a shared conceptualization [16]. Ontologies are different from taxonomies or relation schemas that focus mainly on hierarchical organisation or structural constraints. Ontologies include a semantic layer that encompasses the vocabulary of a domain and logical relationships between concepts. This renders ontologies very flexible to represent complicated and heterogeneous domains such as software supply chains.

The **Web Ontology Language (OWL)** contributes to RDF and RDFS a more expressive collection of constructs for stating more expressive axioms.

OWL supports specifying class equivalence and disjointness, cardinality constraints on properties, properties with features such as transitivity, symmetry, or functionality, and class expressions. By enabling these more explicit semantics, OWL makes it possible for us to perform logical inference on ontologies: a **reasoner** can classify automatically objects, detect inconsistencies, or infer new relations which aren't explicitly asserted.

For pragmatic purposes, OWL has three main profiles (OWL Lite, OWL DL, and OWL Full) of increasing expressiveness and computational expense, and the more recent profiles OWL 2 EL, QL, and RL, intended for large-scale reasoning, query answering, and rule-based inference respectively [21]. This has established OWL as a widespread and de facto standard ontology engineering tool. In security and software development, ontologies have been exploited to model software artifacts, dependencies, and vulnerabilities. Examples include software process management ontologies, vulnerability databases, and security standards. For the Software Bills of Materials (SBOM) example, ontologies allow us to harmonize different standards (e.g., SPDX [34], CycloneDX [32], generic dependencies data) and link them with external knowledge like vulnerability knowledge bases (CVE [12], OSV [29]).

Ontologies are employed as the foundation here to create semantic knowledge graphs that integrate SBOMs, vulnerability data, and open-source project metadata. OWL specifically provides expressive power to capture relationships such as "a weakness affecting a library implies risk for all projects transitively depending upon it". A structured representation such as this supports not just more expressive queries but also automated reasoning, data augmentation, and use of graph learning techniques. Ontologies thus play a twofold function: they represent the conceptual world of CERN's software supply chains as well as provide the interoperability layer required to connect it to external ecosystems.

2.3.3 Knowledge Graphs

A **Knowledge Graph (KG)** is a structured representation of knowledge where entities (nodes) are connected by semantic relationships (edges). Each node typically corresponds to a real-world object or abstract concept, while the edges represent the relations among them, formally defined by an ontology. Knowledge graphs therefore extend the principles of ontologies and the Semantic Web by materializing them as interconnected networks of facts.

A KG can be defined as a directed, labeled graph where the edges represent RDF triples *(subject, predicate, object)*. What distinguishes a KG from a simple RDF dataset is its semantic framework and interconnected structure: a KG organizes diverse data through an ontology or knowledge model, connects entities and relationships into a coherent network, and facilitates reasoning and unified querying. While RDF offers the basic statements, a KG incorporates them into a structured semantic framework that facilitates the integration, navigation, and reasoning of both explicit and implicit knowledge, in particular the one inferred through the logical semantics of RDFS and OWL.

From a functional point of view, knowledge graphs provide:

- **Integration of heterogeneous data sources**, by relying on shared vocabularies and URIs to link data across formats and origins.
- **Semantic enrichment and reasoning**, through the use of ontologies and logic-based inference to derive new relationships or detect inconsistencies.
- **Querying and exploration**, via SPARQL or other graph query languages that allow complex pattern matching over the semantic structure.

By representing SBOMs as RDF data and connecting them to external knowledge bases such as the *Open Source Vulnerability (OSV)* database [29],

the *Common Vulnerabilities and Exposures (CVE)* list [12], or *CHAOSS* community metrics [14], we obtain an enriched and queryable graph of software knowledge. This enables reasoning over indirect dependencies (e.g., transitive vulnerabilities), compliance propagation (e.g., license compatibility across dependency trees), and risk assessment at the ecosystem level.

Knowledge graphs thus constitute the conceptual backbone of this work: they allow SBOMs to be treated not as isolated documents but as interconnected knowledge artifacts, enabling automated reasoning and semantic analysis. The VDGraph project, introduced in the following section, adopts these same principles for integrating SBOMs, vulnerabilities, and open-source project metadata into a unified graph representation—although implemented using a Labeled Property Graph model (2.4.1).

2.4 VDGraph

As described in the previous sections, SBOM standards such as SPDX and CycloneDX offer a detailed and machine-readable description of the components that make up a software project, while **Software Composition Analysis (SCA)** tools such as *OSV-Scanner* [30] report known vulnerabilities affecting those components. These two perspectives remain however separated most of the time, depending on the tools. SBOMs provide the structural and metadata layer of the software supply chain but do not include vulnerability information, whereas SCA tools identify vulnerable components but do not preserve the dependency structure through which those vulnerabilities propagate.

This separation makes it difficult to answer fundamental questions such as:

- Which chain of dependencies links a project to a vulnerable component?
- Are vulnerabilities more likely in direct or transitive dependencies?
- Which vulnerable components are most central or easily reachable in

the dependency graph?

To address this gap, Howell Xia et al. [40] introduced the **Vulnerability–Dependency Graph (VDGraph)**, a unified knowledge representation that merges SBOM and SCA data into a single graph model. The goal is to provide a holistic view of the software supply chain by representing components, projects, and vulnerabilities in a single graph where both dependency relations and vulnerability links are explicitly captured.

2.4.1 Labeled Property Graphs

The implementation of VDGraph is based on the *Labeled Property Graph* (LPG) model, a graph representation in which both nodes and edges may carry one or more labels as well as arbitrary key–value properties, which can be useful to attach different type of dependencies metadata in our case.

An LPG differs from the RDF model used in Semantic Web technologies in several ways. As previously mentioned, RDF represents knowledge as *triples* and all additional information must be encoded as further triples. For instance complex statements such as attaching metadata to an edge (like for dependencies or version constraints), require *reification*, which increases verbosity and complicates querying. On the other hand, in LPGs both nodes and edges can have properties themselves, which allows for more expressive and compact representations.

Neo4j [25], an open source graph database management system, is used for its implementation since it natively supports the LPG model and integrates with the **Cypher** query language, which is optimized for graph pattern matching. These technologies make it possible to easily compute dependency chains, determine the reachability of vulnerabilities, and identify components acting as central propagation points in the supply chain.

VDGraph is constructed as a directed labeled property graph, with nodes of type:

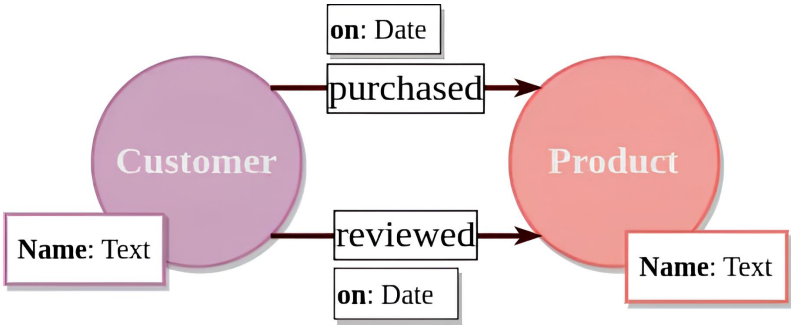

Figure 2.1: LPG example.

- `Root`: represents the underlying software project under analysis.
- `Component`: a software library or component.
- `Vulnerability`: to represent a known vulnerability discovered by the SCA Tool.

Edges represent two types of relations:

- `DEPENDENCY`: a directed edge between components that represents a dependency.
- `HAS_VULNERABILITY`: a directed edge from a component to a vulnerability it is affected by.

2.4.2 Theoretical Properties

Part of the framework includes an algorithm to merge the two data sources (SBOMs and SCA output). This new merged graph has several guarantees to ensure its soundness [40]:

- **Completeness**: all components and vulnerabilities appearing either in the SBOM or in the SCA report are included.
- **Reachability**: every vulnerability is reachable from the root project through at least one dependency path.

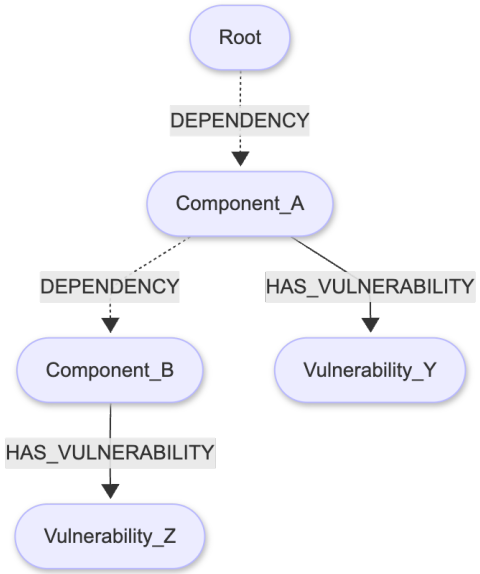

Figure 2.2: VDGraph basic structure.

- **Deterministic Merge**: SBOM and SCA data are combined through a matching strategy based on component name and version, ensuring reproducibility.

2.5 Graph Attention Networks

Recent developments in machine learning have introduced methods able of directly learning from graph-structured data. This technique is known as **Graph Neural Networks (GNNs)**, and they have shown strength in contexts where the relationships among entities are as important as the entities themselves [19].

Other neural network architectures like recurrent or convolutional networks are limited in their ability to represent such structures, while GNNs are conceived to explicitly deal with any type of graph topology. In particular, a GNN updates each node's representation based on its neighbourhood:

$$h'_i = AGG\{f(h_i, h_j) : j \in N(i)\} \tag{2.1}$$

where:

- h_i: feature vector of node i.
- $N(i)$: neighbourhood of node i.
- f: a learnable transformation.
- AGG: an aggregation function such as sum, max pooling, mean, etc...

The main issue with plain GNNs here is that they treat all neighbouring nodes as equally important. However, in our context of code analysis, different functions do not contribute equally to the complete behaviour of the system. Some functions are central to program logic while others could be helpers or can be rarely executed. For this reason, in this study it has been used a variation that gives different importance to different neighbours: **Graph Attention Networks (GATs)**.

This type of GNN, introduced by Veličković et al. [38], overcomes the original limitation by adding to the structure the *attention mechanism*. Instead of uniformly aggregating information from all connected nodes, GATs learn to weight each neighbour based on its relevance to the target node.

A GAT layer is structured as:

- Linear feature transformation: each node's features are projected into a learned latent space using a shared matrix W.
- Attention score computation: for each connected pair of nodes i and j, the model computes a learnable score e_{ij}, representing the relevance of node j to node i.
- Normalization and aggregation: the scores are then normalized with a softmax function to generate attention coefficients α_{ij}, which determine how much influence each neighbour has when updating the representation of i:

$$h'_i = \sigma\Big(\sum_{j \in N(i)} \alpha_{ij} W h_j\Big) \tag{2.2}$$

The coefficients this way computed can be inspected, which gives GATs a certain level of explainability: neighbours assigned with higher weights are interpreted as being more influential.

GATs tipically employ multi-head attention, meaning that several independent attention mechanisms operate in parallel and that their outputs are concatenated or averaged in order to stabilise learning and to allow the model to capture different patterns in the relations.

2.6 Vulnerability Remediation Through Graph-Based Code Analysis

Recent work in software engineering has explored graph-theoretic and learning-based approaches to understand how changes in external dependencies affect the internal structure of software projects [33, 42, 39]. This line of research stems from the observation that dependency upgrades, even when intended to remediate vulnerabilities, can propagate their effects indirectly through the codebase, leading to behavioural changes or breakages [15]. To study these phenomena, the literature increasingly models programs as graphs of functions or control-flow elements and examines how structural properties correlate with the success or failure of remediation attempts.

2.6.1 Function-Level Graphs

A widely used abstraction for representing program structure is the *function-level graph*, where each node corresponds to a function or method and directed edges represent caller–callee relationships or control-flow dependencies [42, 33]. This model expresses how execution is organised internally: some functions centralise substantial coordination, while others operate in narrow or deeply nested regions of the code.

This structural view enables a principled analysis of how external modifications may influence internal behaviour. For instance, if a dependency update alters the semantics of a library function, the change may propagate along the outgoing edges of that node and indirectly affect functions that rely on it. Graph-theoretic features—such as node connectivity, distance to other regions of the graph, or the presence of branching structures—offer a mathematical tool for studying these interactions [3, 13].

2.6.2 Structural Analysis for Vulnerability Remediation

Vera et al. [39] propose analysing dependency remediation by comparing the function-level graphs of different versions of a project, for example the original implementation, an upgraded version that compiles successfully, and an upgraded version that fails. Their analysis reveals that failures often arise in structurally marginal parts of the graph, where functions may not be globally central but are positioned in ways that make them sensitive to upstream changes. In contrast, functions that are more embedded in stable or cohesive regions tend to better absorb dependency updates.

The underlying insight is that the effect of a remediation cannot be assessed solely by examining the functions that directly change. Instead, the structural context in which those functions are located—how many dependencies they have, how they connect to the rest of the program, and how execution flows through them—plays a crucial role in determining whether an upgrade will behave safely or lead to breakage.

In summary, viewing software systems through the lens of function-level graphs provides a theoretical basis for understanding how dependency updates propagate through internal execution structures. Structural properties of these graphs help explain why some remediation attempts integrate smoothly, while others cause unexpected failures. These ideas form the conceptual foundation for the project- level and function-level analyses developed in the next chapter,

where these theoretical insights are applied to real-world software systems.

2.6.3 Graph-Theoretic Metrics for Function-Level Analysis

In this type of function-level analysis, structural properties of the call graph can indicate how specific functions affect the overall behaviour of the application. As updates in external dependencies can influence certain execution paths, centrality and connectivity metrics offer a systematic method to identify which functions are structurally important and are thus more susceptible to changes.

Some of the most relevant measures in this context are [39, 10]:

- **Degree centrality**: measures the number of direct connections a function has. Functions with a high degree frequently serve as entry points, making them sensitive to alterations in surrounding logic.
- **Clonesness centrality**: indicates how near a function is–in terms of shortest paths–to every other function in the graph. A high degree of closeness indicates features that can quickly impact or access numerous areas of the application.
- **Betwenness centrality**: detects functions acting as bridges along execution paths. These nodes frequently mediate control flow between otherwise remote sections of the program and can be essential for grasping the effects of dependency-induced behavioural changes.
- **Clustering coefficient**: used to assess how tightly connected the neighbours of a function are. Local clustering emphasizes areas of closely linked logic, which may be more stable yet also more vulnerable when updates to dependencies impact common routines.
- **Connected components**: assist in evaluating fragmentation in the call graph. Alterations in external libraries can lead to previously connected

components either separating or combining, indicating modifications in accessible functionality.

These metrics offer clear, model-independent indicators of function importance that can be incorporated within our framework along GAT embeddings. Their inclusion allows the analysis to merge domain-agnostic graph structure with application-specific signals indicating which functions are more prone to being impacted during dependency updates, as previously shown in studies on vulnerability remediation via graph analysis.

Chapter 3

Ontologies design

3.1 Competency Questions

One of the main challenges when developing an ontology is determining its scope: what concepts to cover, what level of detail is appropriate, and what kind of questions the ontology will be required to respond to. To guide this process, ontology engineering often employs **competency questions** [17]. These are natural-language, everyday questions that are the candidate use cases for the ontology. They not only provide a design orientation by dictating what concepts and relationships must be expressed, but also an assessment criterion, since the finished ontology can be tested for validity by seeing whether it can answer those questions (typically via SPARQL queries, as in our case).

After discussing with some of the members of CERN OSPO, we concluded that the main questions that the system should answer are:

1. What are the most used libraries/languages/frameworks? Which ones are the most used in a certain department/group?

2. Which components have a certain license or license type? For example, list components having copyleft license.

3. What projects have conflicting licenses? Some CERN projects could be declared as MIT-licensed but depend on GPL-3.0-only, leading to

conflict, so it requires to understand how licenses propagate among the dependencies.

4. Which projects have a certain known vulnerability? For example, which applications are affected by `CVE-2021-44228` ("log4Shell"), either directly or through transitive dependencies?

5. Which projects use a certain library with a certain version? For instance, which CERN services use the Python library `flask` in version `2.3.2`? This is key for supply-chain security use cases, requiring modelling of version ranges, dependency graphs, and version comparison semantics.

6. Which libraries are part of a project but never used? This may indicate unnecessary dependency bloat and opportunities for reducing the project's attack surface

7. Which parts of the codebase are under a certain license and which ones under another?

8. Meta-information such as:

    8.1. Who has been searching for some specific data/queries?

    8.2. Where is the SBOM stored? This is an issue since at CERN there is not a central inventory. It could be DependencyTrack, DejaCode or any other.

    8.3. How has it been uploaded? Through GitLab CI, manually, ...

9. How can service elements be grouped according to the libraries they rely on, e.g. which service elements use `pandas`, which use `numpy`, and which use both? It requires integrating organisational with software usage data.

10. Which tool has been used to generate the SBOM? For instance, which SBOMs in the EN department were produced with `Trivy` and which

with `Syft`? This could give insight on trends inside the organisation worth to investigate.

11. What service elements could be hit by a certain vulnerability? Similar to the CQ about applications affected by vulnerabilities, but in the broader view of service elements.

12. Survey specific (see Section 4.1 and Appendix A):

    - What are the answers to the Survey? Which ones are SBOMs and which not?
    - Who (intended as both persons and organic units) has answered and who hasn't?

The conceptual ground covered by the domain — services and organisational entities at CERN, SBOM standards such as CycloneDX, software licensing, and open-source project metadata — was too vast and heterogeneous to be meaningfully represented in a single monolithic ontology. For this reason, the design was modularized into a number of related ontologies [36], each targeting a distinct conceptual domain: the CERN Ontology for organisational and service entities (Section 3.2), the CycloneDX Ontology (Section 3.3), the SPDX Ontology (Section 3.4), the Software Licenses Ontology for license types and their interrelationships (Section 3.5), and the OSPO Ontology for open-source program office activities and project metadata (Section 3.6). This modular approach simplifies maintenance and promotes reuse but still allows integration into a unified knowledge graph that maintains the competency questions previously defined.

3.2 CERN Ontology

One of the ontologies developed from scratch for our case is the CERN ontology. It captures the outlined organisational structure and the current taxonomy of software services as they currently are in their internal service catalogue, as explained in Section 2.1.2. Organisational units are captured through `OrgUnit` and its specializations (`Department`, `Group`, `Section`). Leadership of organisational units are generically represented as `Leader`, since they are only needed to keep track of for their contacts there is no need for major granularity. More in the detail, the structure is:

$$\{\texttt{Leader}, \texttt{ServiceResponsible}\} \sqsubseteq \texttt{Person},$$

$$\texttt{Person} \sqsubseteq \texttt{EmailHolders}$$

The only other entities defined as subclasses of `EmailHolders` are `EGroup` and `ServiceAccount`, the former being only a generic email that can be used as contact point for services and the latter being a user account representing a non-human entity.

The service catalogue is captured through `ServiceCatalogComponent` with subclasses:

- `ServiceArea`
- `CustomerArea`
- `ServiceElement`
- `FunctionalElement`

Structural relationships follow a decomposition pattern via `hasPart` (specialized as `hasServiceElement` and `hasCustomerService`) to assert the way elements and regions make up the catalogue.

The primary objective of the ontology is to provide rich organisational context for SBOMs produced in many units, together with services and components, matching up with the units and people responsible for them so that downstream analysis (mainly sharing dependencies, licenses management, and distribution of known vulnerabilities across services) can be accomplished consistently.

3.2.1 Design patterns

Besides the *part-whole pattern* already described for the services taxonomy and the *hierarchy pattern* described for the organisational units and email holders, the other design patterns used for this ontology are:

- **Membership pattern**: `memberOf` links `Person` to a `OrgUnit`, enabling the possibility to ask questions such as "Which people belong to a certain unit?".
- **Participation pattern** via subproperties. Bidirectional "support" relations between functional and service elements are modeled with `supports` (domain `FunctionalElement`, range `ServiceElement`) and `supportedBy` (domain `ServiceElement`, range `FunctionalElement`) as it is on the services platform. Granularity is captured by specialized subproperties (`supportedByPrimary`, `supportedByRegular`, and `supportedByOccasional`) to qualify the mode or intensity of support without introducing reified n-ary events.
- **Communication pattern**: contactability is factored via `EmailHolders` with the datatype property `email`, and its specialized subclasses as `EGroup` and `ServiceAccount`. This provides a unique hook for notifications tied to SBOM findings.

3.2.2 Covered Competency Questions

This ontology contributes to the overall knowledge model by enabling the formulation and answering of a set of competency questions mainly related to software governance and vulnerability management. Some examples include:

1. **Software usage and technology landscape**: it helps outlining the most used libraries, frameworks or languages throughout the organisation and specifically which ones are more predominant in each single *department* and *group*.

2. **Vulnerability management**: which *applications* are affected by a certain known vulnerability, and which *service elements* could be indirectly impacted through shared dependencies?

3. **Data aggregation and impact analysis for services**: how can *service elements* be grouped based on the libraries they use, and which SBOM generation tools they use? What *service elements* could potentially be affected by the spread of a particular vulnerability?

3.3 CycloneDX Ontology

As previously mentioned, CycloneDX is only defined by an XML schema with no OWL ontology for it, as SPDX. For this reason, it was necessary to adapt it for our case. Given the complexity of the standard, only part of it has been translated. The chosen reference version is `CycloneDX v1.4` since it is the first to include free properties for components, needed in our workflow, and all the successive iterations are backward-compatible with it. An interesting feature of CycloneDX is the possibility to focus the *Bill Of Materials* on different domains such as machine learning (MBOM) and cryptography (CBOM), but in our case we will only work with the *software* bill of material.

The only classes identified as in scope were:

- `SBOM`: it represents any CycloneDX SBOM that is loaded into the ABox.
- `Component`: one to one mapping of the elements in the SBOM component's list. It is at the core of the ontology.
- `Hash`: in the standard it can be attached to almost any entity, in our case it is mainly used for Components and Tools. Each instance is defined as being equivalent to exactly one `hashAlgorithm` and one `hashValue` (stored simply as `xsd:strings`).
- `Tool`: it represents the tool with which the SBOM has been generated. Example instances are *Trivy*, *Syft*, *Scancode*, ...

The main object properties for the ontology are:

- `hasComponent`: associates an `SBOM` to a `Component`.
- `dependsOn`: property for mapping the `dependencies` array of the standard. It associates `Components` among them.
- `hasHash`: associates any type of entity to a `Hash`.
- `createdWith`: associates an `SBOM` to a specific `Tool`.

This ontology then includes several data properties. The main ones being `purl`, `vcsReference`, `hashValue` and `hashAlgorithm` to be associated with `Hash` instances, metadata of the SBOM like `serialNumber`, `timestamp` and so on.

3.3.1 Covered Competency Questions

This ontology, although only limited to projects components and SBOMs artifacts, helps answering some of the CQs, such as:

- **Most used libraries/languages/frameworks and projects using a specific library/version**: `Component` allows to extract all the libraries declared in CycloneDX SBOMs, which supports frequency analysis across units.
- **Libraries present but unused (partial)**: By enumerating components in the project, the ontology provides the baseline for comparing the inclusions with function-level usage.
- **Grouping service elements by libraries**: together with the `cern` ontology it enables grouping and comparison of libraries by service elements.
- **Which tools generated the SBOMs**: with `Tool` and `createdWith` the query is straightforward.

Throughout this thesis, this ontology will be referred to with the prefix "**cdx:**".

3.4 SPDX Ontology

Unlike CycloneDX, the SPDX standard includes an official OWL ontology as part of its ISO/IEC 5962:2021 specification. Consequently, the ontology utilized in this study does not modify SPDX concepts but rather selectively reuses useful classes and properties from the official vocabulary. The aim is to encapsulate the same conceptual framework applied for CycloneDX—software elements, hashes, and tools—in a manner that enables the OSPO ontology (Section 3.6) to uniformly generalise both standards

For our scope the classes identified as useful were:

- `SpdxDocument`, which simply represents an SPDX SBOM. It is linked to a `CreationInfo` with a `creationInfo` object property and to multiple `Packages` via a `spdx:describes`, which is the analogue of `cdx:hasComponent`.

- `Package`, the main class used to represent software components in the standard. It is conceptually equivalent to `cdx:Component`. It can, and should, be linked to a `Checksum` with a `checksum` property.
- `Checksum`, represents cryptographic hashes and it serves the purpose of identifying `Pacakges` uniquely.
- `CreationInfo`, used for describing SBOM generation metadata, including creator, timestamp and tool used for generation.

No other relevant classes or object properties are reused from the standard. In particular, detailed SPDX licensing and file-level semantics are intentionally left to the Software License Ontology (Section 3.5) and OSPO ontology (Section 3.6), keeping the SPDX layer focused on components and metadata only.

Considering their similarities, also the CQs answered by this ontology are analogue to the ones answered by the CycloneDX one.

3.5 Software License Ontology

Given that software licensing is a well-established domain within software engineering, this work relies on an existing and widely adopted model rather than introducing a new one from scratch. In particular it integrates the **Software License Ontology** [41], developed by researchers at the University of Stuttgart in support to the License Checker tool they also developed, that aims at supporting software developers in choosing the right license for their code base.

Since its only scope is to handle licenses it is a simple ontology, with the only classes being `Condition`, `LicenseType`, `Limitation`, `Permission`, `Recommendation` and finally `SoftwareLicense` as subclass of `spdx:License`.

There are also few object and data properties. The ones that interest us the most are `hasPermission`, `hasLimitation`, `hasLicenseType`,

`isBackwardCompatibleWith`, `isForwardCompatible`, `isOsiApproved`, `isFsfLibre` and `url`.

The ontology includes also several instances of the mostly used licenses like *GPL 3.0*, *MIT* and *Apache 2.0*, which makes it a good fit for plugging it in inside ours.

In what follows, this ontology will be referred to with the prefix "**slo:**".

3.6 OSPO Ontology

All previously defined ontologies cover only certain parts of the context and help answer some specific competency questions, but it is still missing something that can put them together in order to have a complete overview of the information collected.

Here is where comes into play the OSPO ontology, referred to with the prefix "**ospo:**", that is intended to glue the `cern`, `slo`, `cdx` and `spdx` ontologies – and potentially any other SBOM standard – together in order to answer all competency questions and give a tool for OSPO members to ease part of their workflows.

The main idea for this ontology is to generalise all others in order to have a standard framework to look at the data coming from different sources and formats, or even in plain text form as in the dependency survey (appendix A).

First of all this ontology generalises the concept of SBOM with simply `ospo:SBOM` that becomes a superclass for `cdx:SBOM` and `spdx:SBOM`. For all `ospo:SBOM` there is a one to one relation (`ospo:hasSBOM`) with an `ospo:Software`, which is a representation of any kind of software entity. It gathers as child the classes `cern:Application` and `ospo:SoftwarePackage` –the latter having `ospo:SoftwareVersion` as further subclass–where:

- `cern:Application`: represents any application running, developed or maintained at CERN.

- `ospo:SoftwarePackage`: represents a generic software unit with no ties to any SBOM and in particular with no information about versions. For example it could be Flask, Pandas, Numpy, ...

- `ospo:SoftwareVersion`: represents a specific `ospo:SoftwarePackage` with associated a specific version number (`ospo:version`). It is also a generalisation of SBOMs components, so more specifically of `cdx:Component` and `spdx:Package`. For example it could be "Pandas v2.3.3".

In this ontology it has also been employed the `SKOS` (Simple Knowledge Organization System) vocabulary [5] to represent the concept of SBOM type. Unlike the distinction between SPDX and CycloneDX, which is expressed at the class level, SBOM types such as *build*, *source* or *runtime* are modelled as instances of a `skos:ConceptScheme`. Each type is a `skos:Concept` within this scheme, providing a controlled and extensible terminology for classifying the purpose or generation context of an SBOM. Individual `ospo:SBOM` instances are linked to their corresponding type using the property `ospo:sbomType`. This approach keeps the class hierarchy lightweight while allowing OSPO members to distinguish and query SBOMs according to their role in the software lifecycle and allows the usage of some other interesting SKOS properties like `skos:broader` and `skos:narrower` that can be used among different versions of SBOMs.

Vulnerabilities are represented with a `ospo:Vulnerability` class, that has its main relation (`ospo:hasCVE`) with a `ospo:CVE`, containing formal identifier and associated metadata. The impact of a vulnerability is not expressed directly on software versions, instead it is used the `ospo:VulnAffects` reification for capturing the affected packages and the version ranges. This way, an `ospo:SoftwareComponent` can be connected to the vulnerability it is affected by, together with information of versions impacted (`ospo:introducedVersion`, `ospo:lastAffectedVersion`, `ospo:fixedVersion`).

In order to answer the CQs regarding the dependency survey (Section 4.1), which will be a recurring thing during the years, it has been added some classes to properly describe it:

- `ospo:Survey` represents a single survey. It has a name (`dcterms:title`), a description (`dcterms:description`) and several answers associated to it with `ospo:hasAnswer`.
- `ospo:SurveyAnswer` represents a single answer to the survey and it has two subclasses: `ospo:SBOMAnswer` (equivalent to an answer having associated to it some `ospo:SBOMs`) and `ospo:TextAnswer`, this latter to represent a manually filled in form answer.

It has also been represented the concept of agents and events, with `ospo:Agent` and `ospo:Event`. An agent can be anyone uploading to the survey, OSPO or computer security team members making queries to the knowledge base. On the other hand, `ospo:Event` has some subclasses:

- `ospo:QueryEvent`: represents the event of a query being made to the KB by a certain entity.
- `ospo:UploadEvent`: it represents the uploading of an SBOM somewhere with `ospo:uploadedTo`, by someone (`ospo:uploadedBy`). It has two subclasses more:
  - `ospo:SurveySubmissionEvent` is the submission of an answer to the survey with all the metadata associated with it, like the `cern:Person` doing it and the `xsd:dateTimeStamp`. It corresponds to an `ospo:UploadEvent` with `ospo:uploadedTo` equal to `:Survey`.
  - `ospo:SystemUploadEvent`, which represents the storage of an SBOM to an inventory. Since there is not a single one across CERN it helps keeping track on where the SBOMs are being stored.

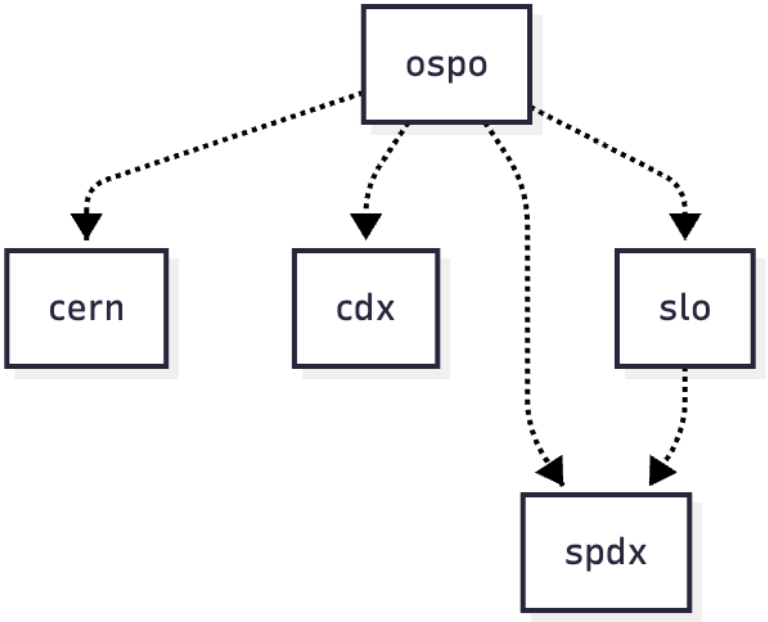

Figure 3.1: Ontologies usage.

Other relevant properties are those that link together the different ontologies. Some worth mentioning are:

- `ospo:softwareUsage`, which is a generalisation of `ospo:FEusesSW` and `ospo:SEusesSW` to link respectively `cern:FunctionalElements` and `cern:ServiceElements` to an `ospo:Software`.
- `ospo:hasLicense` that binds any `ospo:Software` to an `slo:SoftwareLicense`.
- `ospo:dependsOn` which is a generalisation of components dependencies (so of `cdx:dependsOn` and `spdx:dependsOn`) that is also paired by its inverse property `ospo:isDependencyOf`.

3.6.1 Design patterns

First of all, the clearest pattern that is also behind the whole idea of this ontology is the modularization based on the different contexts using different vocabularies, the ones described above in this same chapter.

Considering that the primary focus of this ontology is generalisation, a natural pattern to be used is simply the hierarchy one. For example it introduces classes such as `ospo:Software` and `ospo:SoftwareComponent` that generalise the other ontologies concepts of components or `ospo:SBOM` with the same intent. Besides this, it has also been employed in several other representations, such as the ones for events and survey answers.

Since one of the limitations of OWL is that it does not allows to have labels or properties in the relationships themselves, for representing the vulnerabilities affecting components it was necessary to use the n-ary relation pattern (reification) for them. This way the single components are not connected directly to a vulnerability, but to an instance of this intermediate class `ospo:VulnAffects` that gathers all the information regarding a certain vulnerability affecting that specific package.

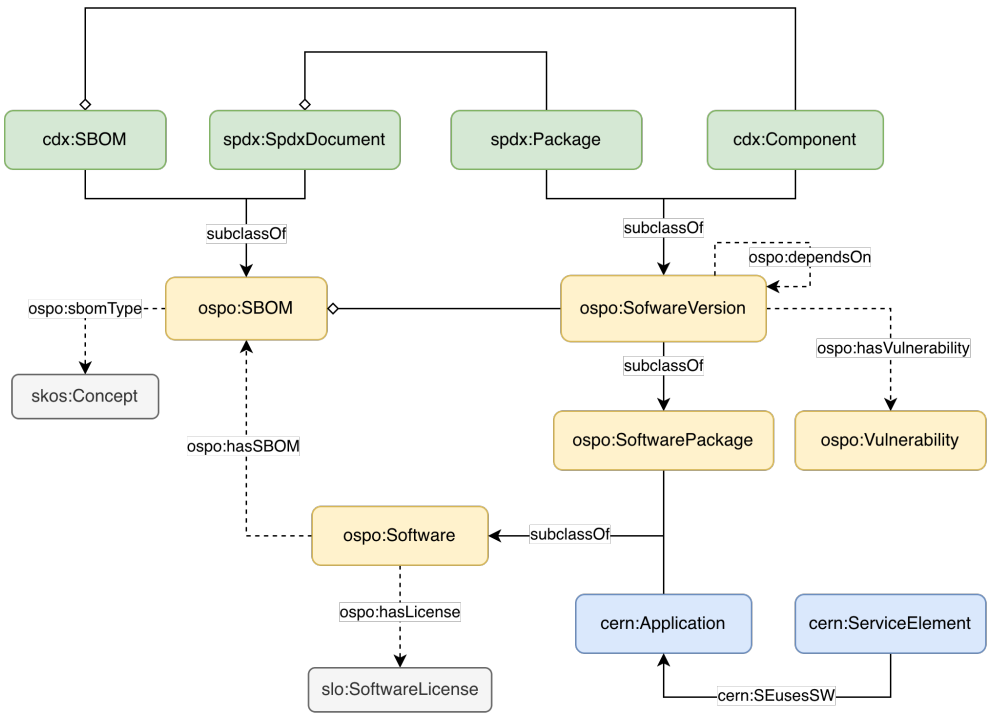

Figure 3.2: High-level conceptual overview of the core ontology classes (UML-inspired diagram).

3.6.2 Covered Competency Questions

This ontology serves the purpose of attaching together all other ontologies while adding some other concepts that were OSPO-specific. This convergence makes it able to answer nearly all the competency questions defined during the design phase, allowing for organisational and security-related queries coherent and expressible through a single semantic framework.

In Figure 3.2 it can be seen an overview of the main classes of the final ontology and some of their relations.

Chapter 4

Methodology

4.1 Dataset

At CERN, a main concern is that there is no organisation-wide overview of the software ecosystems among the diverse units. For instance in some departments, like Accelerator Beams, the languages used in developing internal tools are only Java and Python, which helps in defining standard procedures and workflows for securing the software supply chain. This department in particular is the only one having a DependencyTrack instance - a storage system for CycloneDX SBOMs - set up and running that is constantly updated during the software lifecycle that is being developed by them. Besides this organisational unit all other have a too varied ecosystem of languages that impedes the definition of standards in this context.

As a starting point towards having a CERN-wide perspective of what is running inside the organisation, the Computer Security Team and the OSPO have decided to create a survey, named ***CERN Open-source Software Dependency Survey***. As a first step group leaders have been asked to declare by email the applications or services under their responsibility and at a later stage to fill the actual dependency survey set up on Google Forms. It has been decided to send out the survey to the main units that develop software, which are:

- IT Department (Information Technologies).
- BE Department (Accelerator Beams).
- TE Department (Accelerator Technology).
- EN Department (Engineering).
- SY Department (Accelerator Systems).
- FAP-BC Group (Business Computing).
- EP-SFT Group (Software Design for Experiments).

In this survey, better described in Appendix A, the service managers were asked to declare the applications used by their services and the main packages used by them. They could either fill the form manually in text format or upload SBOMs for each application, adding to them some CERN-specific metadata (according with the SBOM standard) with a Python tool was developed. These extra properties were:

- Official application's name
- CERN organisational unit
- Service element to which it belongs.
- Email of the service manager.
- Official Version Control System (VCS) URL of the application's repository.

Besides SBOMs and the survey, in order to have a more holistic understanding, the data regarding these applications has been complemented with the **Abacus data**, the application used by CERN to catalogue and manage IT services and applications, including their ownership, functional roles, and dependencies. This data includes all the service elements with their relationships

and the most important IT applications with the high-level degree of dependency among them, in order:

- *Supportive*: if the dependency is broken it does not directly impact the main application.
- *Semi*: the breakage of the dependency influences only partially the main component.
- *Full*: while the dependency is down, the main application will be down too.

This information can help better understand how much certain vulnerabilities can impact not only a single application but the whole ecosystem of services present in the organisation.

Raw Data Analysis

After the deadline for replying to the survey, we received:

- 26 enriched SBOMs, all in CycloneDX.
- 19 text answers to the survey.
- 8 answers by email, with survey-like CSV files.

Among the 26 SBOM answers there are two corner cases. One is the FAP-BC response, where they uploaded a single SBOM for the their whole Bitbucket instance with no distinction among repositories. This gives an overview of the pacakges used but it is not very meaningful, for this reason it has been excluded from the general analysis even if it could be queried alone to still get some insights. The second case is the one for the LanDB software where it has been submitted a single SBOM for each module of the application. This is good and it should be the recommended, but it is not good for our study case since its 96 SBOMs would bias too much the results.

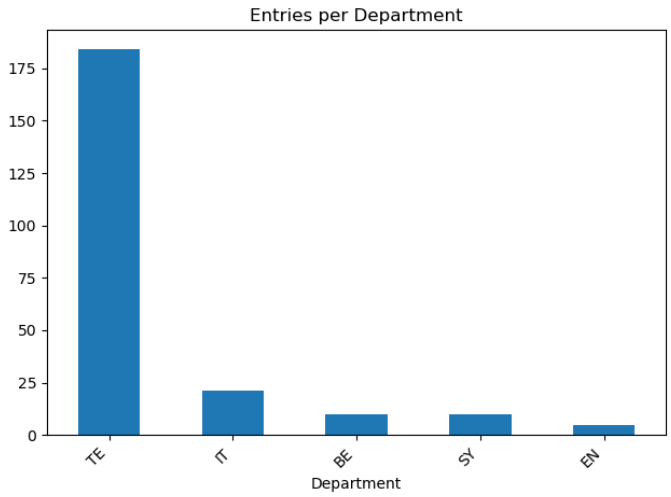

Figure 4.1: Number of declared software per unit.

Figure 4.1 illustrates the number of entries (meaning either applications or services) declared, grouped per organisational units. The dataset is primarily influenced by contributions from the TE department, which provided over 180 entries. Other units such as IT, BE, SY and EN, show lower yet still significant numbers. This disparity is already interesting to see, since it can show the disparity in SBOM and well practices adoption.

Another interesting thing is the number of components per SBOM, which can be seen from Figure 4.2. The majority of SBOMs contain fewer than 250 components while some applications demonstrate much larger dependency sets, exceeding one thousand components. These larger SBOMs usually relate to more intricate systems, integrated services containing various plugins, or applications that include complete dependency trees.

Similarly, we can also check the most common ecosystem derived from component PURLs. Apparently, from 4.3, the most common ecosystems are npm and Maven, succeeded by Cargo an PyPi, indicating diverse and multi-language landscape.

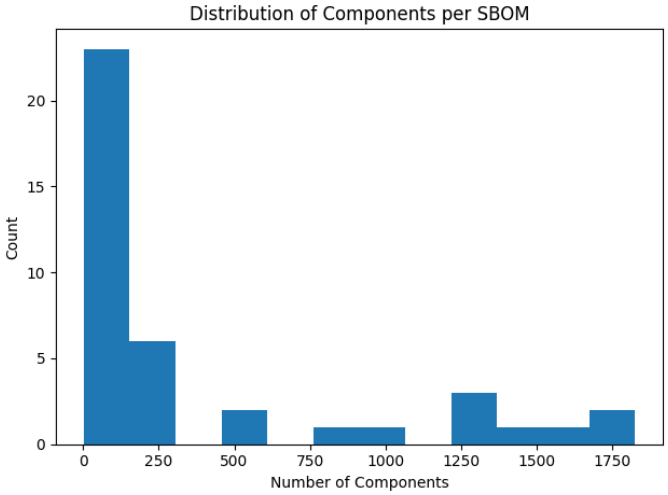

Figure 4.2: Number of components in SBOMs.

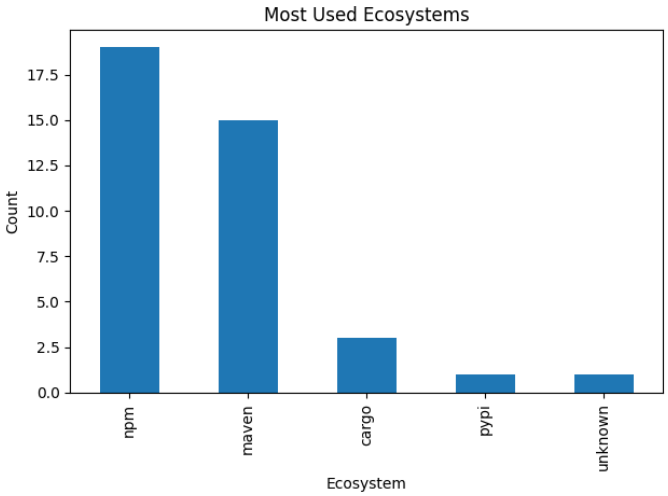

Figure 4.3: Ecosystem usage among submitted SBOMs.

4.2 Pipeline

During my internship at CERN I developed several tools that implemented all the previously described technologies separately. They can be orchestrated to work well together in a specific workflow.

4.2.1 Survey2RDF for data conversion

The survey answers, both SBOMs and CSV files, are stored in a repository in CERN's internal GitLab instance. The data can be downloaded and it can be processed with a Python tool called simply `Survey-Analysis`, that is mainly based on `rdflib` library and it is composed of several modules. The first one is `Survey2RDF`, that given either an SBOM or a CSV file structured according to the survey format outputs a complete ABox in Turtle RDF using and following the ontologies previously described. Inside this tool there is also a utility that converts the Abacus data to RDF as well, which has been used for this study but opposite to the survey data it is not available to everyone.

4.2.2 LIMES for data linkage

Once all datasets have been converted into RDF, the next step in the pipeline consists of linking semantically equivalent entities across the different sources. For this purpose, the workflow leverages **LIMES (Link Discovery Framework for Metric Spaces)** [26], a widely used tool in the field for efficient link discovery in knowledge graphs. LIMES is particularly suitable for this scenario because it optimizes similarity computations by exploiting metric properties and approximation techniques, thus reducing the overall computational cost [27].

It operates on declarative link specifications, which define how entities from two RDF datasets should be compared, which similarity measures should be applied, and which thresholds determine equivalence. These specifications

are expressed in the LIMES specification language (LSL), allowing both expressiveness and reproducibility [27].

In this project it has been used for two purposes. First of all, for **matching survey applications to Abacus applications**. Since the names of the applications in the survey are declared by humans they may have errors, abbreviations or bad formatting and hence they could not match Abacus data by default. Besides this, it is also used to **detect equivalent components across SBOMs**. Since they are generated by different tools they may follow different conventions when declaring IDs, PURLs, version strings and so on. Discovering which found components reference the same entity can give interesting insights and enables more meaningful queries on the whole landscape.

For both these cases it has been used the Levenshtein distance to discover links. It is a classic string similarity metric that measures the minimum number of single-character edits—insertions, deletions, or substitutions—required to transform one string into another [24]. If the similarity is greater than a certain threshold A and below another threshold B (different in the two cases) it is created a triple relating the two objects with the `limes:near` relationship, while if it is greater than threshold B they are associated with `owl:sameAs`. The first output is omitted since it would required manual check of large data, but the second is automatically added to the ABox and allows to interpret the entities as representing the same concept after the reasoner run.

4.2.3 Ontology Query Engine

After all the data has been correctly converted, the user (OSPO or computer security team member) can query it with the main module of the tool, called `OntologyQueryEngine`, through a CLI. It can run an OWL RL reasoner to find new relationships among the triples and link the LIMES findings. The user can execute predefined queries like the competency questions, execute debug ones like “get all relationships/entities of a certain type”, “list all SBOMs”

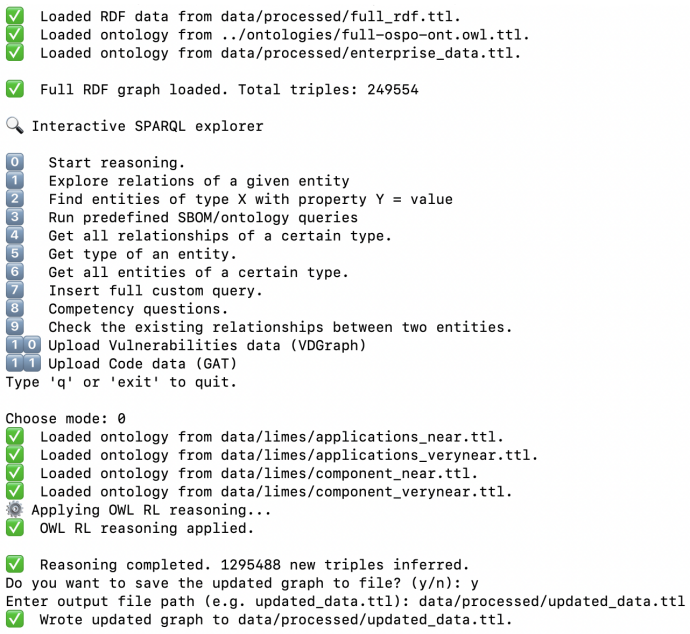

Figure 4.4: Ontology Query Engine CLI.

or write a personalised query. Some of its queries will be illustrated in Section 4.3.1.

4.2.4 VDGraph

Either from the CLI or from the standalone tool `VDGraph` the user can load an SBOM to a local instance of Neo4j structuring the data as specified in Section 2.4.1. Many SBOM generation tools only report direct dependencies so first of all, in order to populate the data with transitive dependencies too, the SBOM is scanned and for each component it describes the tool queries `deps.dev`, an API provided by Google that gives metadata about open-source packages across many ecosystems. As a second step, for each direct and transitive dependency it is also queried `OSV` [29], an open vulnerability database designed specifically for open-source ecosystems, in order to get all components' vulnerabilities. While these data about vulnerabilities are retrieved and uploaded to Neo4j, they are also converted to RDF so it can be later added and queried with the main ontology. In particular the `OSV` information is converted

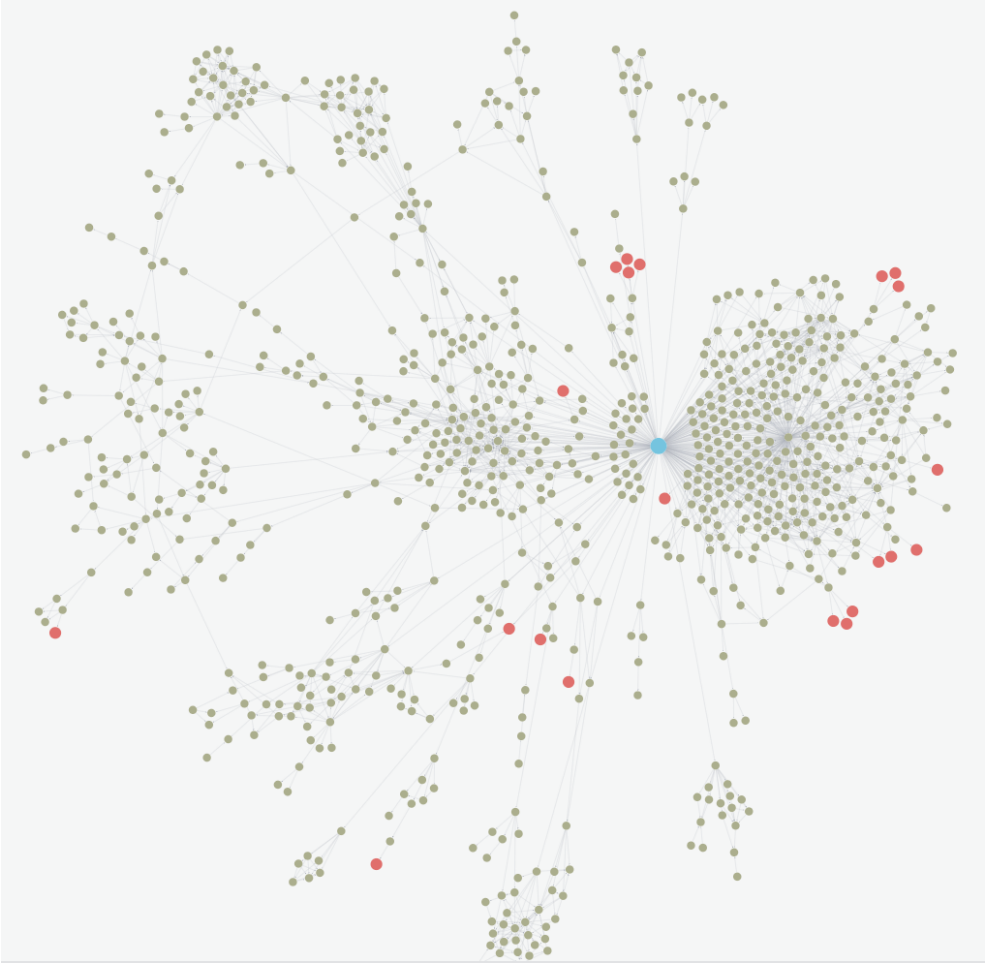

Figure 4.5: VDGraph representation of Indico on Neo4j.
The blue node is the root, red nodes are vulnerabilities, and green nodes are components.

to `ospo:Vulnerability` together with all its other surrounding classes and properties as described in Section 3.6.

Similarly to the Ontology Query Engine, the data can be queried through either custom Cypher queries or the predefined ones, some of which are introduced in the next Section 4.3.2. In addition, Neo4j provides an interactive visual interface that allows users to explore the dependency graph directly, facilitating the inspection of component relationships and vulnerability propagation paths, as it can be seen in Figure 4.5.

4.2.5 Function graph analysis

The final step towards looking inside a project is done via the `Function Graph` tool, that uses node centrality metrics and GAT scores to study a codebase at the function level. While the previous steps focused on having insights at an organisation and project level, this step aims instead at "zooming in" further into a single application. The goal here is to determine which functions are more important from a structural point of view, how they are interconnected and how their position in the call graph may amplify or reduce the effects of dependency updates intended for vulnerability remediation, as explained in Section 2.6. This part of the project is only at an experimental phase, already helping answer some questions but with large margin of improvement. For example it currently works only with Python and Java projects but it could be easily extended to other languages.

In practice, it has been developed a Python tool that builds a directed ***function-call graph*** from the project's source code, which needs to be cloned locally (the VCS URL should be present in SBOMs). The implementation combines static code parsing using the `Tree-Sitter` parsing framework with Git-based differencing to identify changes across releases. `Tree-Sitter` allows to extract from code several features which are finally used to build a graph where each node represent a function and the directed edges represent caller-callee relationships. Per each function the main features extracted are:

- File path and function name, transformed into a unique ID.
- Position in the source file.
- List of called functions.
- Imported modules.

The latter is particularly important in our context since it allows for mapping the individual functions to the specific libraries they rely on, and if matched

with `OSV` and SBOM data this can give hints on which parts of the code base could be affected by a known vulnerability.

After collecting the data and constructing the graph, the tool computes a series of graphic-theoretic metrics: **degree, closeness, betwenness, clustering coefficient** and a global approximation of **cyclomatic complexity** [39], which help characterise the structural importance of each function. In addition to this, it is possible to query the `git` history between two specified tags or commits and the tool determines if the function was modified or not between the two versions. This enables comparative studies of "changed" versus "unchanged" functions, that can reveal how structural positions influence the impact of dependency updates, similar to the claims made by Vera et al. [39] (see Section 2.6).

To complement these metrics, the graph is also analysed using a GAT (Section 2.5). The goal is not to perform supervised prediction but to obtain **node embeddings** that reflect the structural relevance of each function within the program's architecture. The GAT learns representations by assigning different attention weights to neighbouring nodes during message passing, which enables the model to understand which callers or callees are the most influential when updating a function's embedding, performing a sort of learned structural centrality. After training, the vectors represent the functions in an embedded space, and its norm can be treated as a measure of relevance to structure, denoted by its GAT score. From a practical standpoint, functions with high GAT scores are generally architectural entry points, orchestrators of various operations or nodes residing in topologically dense neighbourhoods.

Besides its main focus, during the generation of the function graph it is also possible to trigger the generation of RDF triples that is specific to code units. In particular, at the moment, the following triples are generated:

```
ospo:codeunit-id-N rdf:type ospo:CodeUnit .
```

```
   ospo:codeunit-id-N ospo:hasLicense slo:{LicenseName} .

ospo:codeunit-id-N ospo:hasCopyright {copyright_statement} .
```

Which helps answering some other Competency Questions (Section 3.1) regarding licenses usage that couldn't be answered otherwise. If run for all repositories it could make increase excessively the number of triples, so contrary to vulnerability data that it would be always useful to track, this should be considered only if needed.

4.3 Evaluation and Case Studies

This section evaluates the proposed framework through a set of case studies that showcase its ability to integrate and analyse software-related information at different levels of abstraction. In particular, we illustrate how organisational data, dependency-level information, and function-level code properties can be integrated to answer CQ. Figure 4.6 summarises the overall architecture of the system, highlighting the main components (Survey2RDF, the ontology-based knowledge graph, VDGraph, and the Function Graph Analysis tool), their interactions, and the external data sources they rely on. The following subsections present concrete analyses built on top of this architecture, highlighting both the expressiveness of the underlying ontologies and the practical usefulness of the implemented tooling.

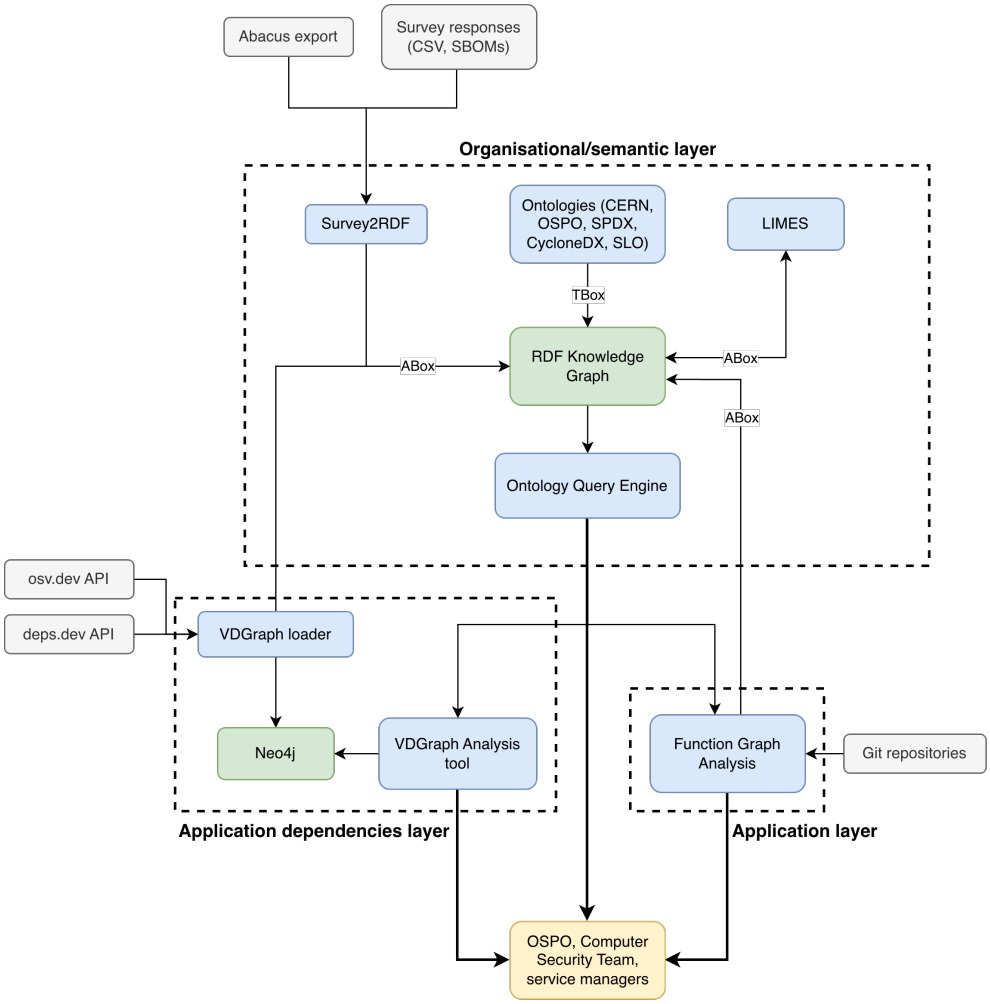

Figure 4.6: General architecture of the framework.
Grey boxes denote external data sources and APIs, blue boxes represent components and analysis tools developed for the framework, green boxes indicate graph data stores while the yellow box denotes the users.

4.3.1 SPARQL Queries

The SPARQL layer is the initial stage for performing analyses on the knowledge graph created from SBOMs, survey and Abacus data. After the data conversion and data linkage introduced in sections 4.2.1 and 4.2.2, the Ontology Query Engine offers support for interactive exploration on the resulting RDF graph using either preset queries representing the CQ initially specified during

ontology design, or general-purpose SPARQL queries specified manually by the user. The purpose of the tool is to serve as a single entry point for querying the different sources including SBOMs, responses to surveys, Abacus data and entity links derived by LIMES. Thanks to the modular ontologies (`cern`, `cdx`, `slo`, `spdx`, `ospo`) all the sources are represented under a shared semantic model, allowing the user to query the data without the necessity of knowing the data origin or format. To demonstrate the type of information that can be extracted from this type of data, a few representative SPARQL queries will be reported in the following.

A question that an OSPO officer could be interested in running is **how SBOMs are generated across different units**, to understand the tooling usage in the organisation. Thanks to the modeling (via the `cdx:createdWith` and `ospo:createdByUnit` properties), this information can be easily computed through a single SPARQL query:

```
SELECT ?sbom ?toolName ?toolVersion ?appName ?org
WHERE {
  ?sbom cdx:createdWith ?tool .
  ?tool dcterms:title ?toolName .
  ?app ospo:hasSBOM ?sbom ;
       dcterms:title ?appName .
  ?sbom ospo:createdByUnit ?org .
  OPTIONAL { ?tool ospo:toolVersion ?toolVersion . }
}
```

Listing 4.1: Query to retrieve SBOM tools per organisational unit

A subset of the output is shown in Table 4.1. From it, it is possible to observe how some units are already using the same tool consistently while some other no, meaning they may require support in adopting standard workflows.

A second interesting analysis focuses on the **relationship between service elements and the applications they depend upon**, using the Abacus metadata merged into the ontology. This allows the Computer Security Team and

org	appName	toolName	toolVersion
cern:EN-EL-CBS	Web Energy Frontend	trivy	0.64.1
cern:TE-MPE-CB	EDAQ Network	syft	1.33.0
cern:TE-MPE-CB	PM Browser	syft	1.33.0
cern:IT-SD-GSS	CEPH	trivy	0.64.1
cern:EN-ACE	dmu-backend	cdx-maven	2.9.1
cern:IT-CA-CTE	webcast-website	GitLab	18.3.5
cern:EN-EL-CBS	Supply Manager Frontend	trivy	0.64.1
cern:EN-EL-CBS	Fibre Store Frontend	trivy	0.64.1
cern:EN-EL-CBS	Tech Panels Frontend	trivy	0.65.0
cern:EN-EL-CBS	Gesmar Frontend	trivy	0.64.1
cern:IT-CA-CTE	Indico	trivy	0.64.0
cern:EN-ACE	team-viewer-front-end	cli	11.5.2

Table 4.1: SBOMs with the tool used to generate them.
The column with SBOM URIs has been removed for better readability.

OSPO to determine which SBOMs correspond to which service-level components – helpful in case of new known vulnerabilities – and to assess coverage of the survey dataset. The SPARQL query is the following:

```
SELECT DISTINCT ?sbom ?appName ?serviceElementName
WHERE {
  ?serviceElement a cern:ServiceElement ;
                  ospo:SEusesSW ?app .
  ?app ospo:hasSBOM ?sbom ;
       dcterms:title ?appName .
  ?serviceElement dcterms:title ?serviceElementName .
}
```

Listing 4.2: Query to retrieve applications and their SBOMs for each service element.

This query provides a mapping between services and the software artifacts

they rely on, and it can be further extended for analysis of vulnerability propagation or license compliance across the whole service catalogue. From Table 4.2 it can be seen how certain service elements are associated with multiple SBOMs while other to only a single application.

sbom	**appName**	**serviceElementName**
su:sbom-survey-136	CERNBox	CERNBox Service
su:sbom-survey-130	CVMFS	CVMFS
su:sbom-survey-152	Apache ActiveMQ	Messaging Service
su:sbom-survey-156	FluentBit	Monitoring Service
su:sbom-survey-153	Grafana	Monitoring Service
su:sbom-survey-154	Mimir	Monitoring Service
su:sbom-survey-149	OpenStack	Server Provisioning Service
su:sbom-survey-137	Authentication (SSO/Keycloak)	Single Sign On and Account Management
su:sbom-survey-138	Identity and Account Management	Single Sign On and Account Management
su:sbom-survey-139	WLCG IAM	Single Sign On and Account Management
su:sbom-survey-187	Mattermost	Mattermost Service
su:sbom-survey-125	Panoramas	Panoramas Service

Table 4.2: Some SBOMs collected from the survey

It is possible to pose many other questions, some other predefined queries include:

- Get all SBOMs (including the design ones from the survey) and their metadata.
- Get all software with a certain license, or with a certain license type (for example copyleft).
- Get all service elements affected by a certain vulnerability.
- Get the most used libraries used by certain organisational units.

Together, these examples demonstrate the practical utility of the semantic integration effort: questions that would require manual reconciliation across multiple spreadsheets, SBOM documents, and internal systems become expressible as simple SPARQL queries over a single knowledge graph.

4.3.2 Cypher queries

As mentioned in section 4.2.4, a project's SBOM at a time can be loaded to Neo4j and augmented with transitive dependency and vulnerability data in order to inspect its internal structure with Cypher queries. While the SPARQL queries give an organisation-level view of the entire semantic knowledge graph, this tool focuses on the structure of a single project's dependency graph. This method allows to answer questions related to vulnerability propagation, dependency depth and to identify structurally important components, similarly to the approach proposed in the original paper of VDGraph [40].

In the following, a few illustrative Cypher queries will be shown. The first two come from the VDGraph paper while the others are part of the new ones added to the tool and useful in our specific context.

Query 1: Path counts

In this first query, it is measured for each vulnerable component, how many distinct dependency paths connect it to the project root. If they are reachable from a large number of independent paths then they represent **critical risk points**, since they allow a vulnerability to spread easily throughout the system.

```
MATCH path = (r:Root)-[:DEPENDENCY*]->(c:Component)
WHERE EXISTS {
  MATCH (c)-[:HAS_VULNERABILITY]->(v:Vulnerability)
  WHERE v.severity = 'HIGH'
}
```

```
RETURN c.name AS component, count(path) AS numPaths
ORDER BY numPaths DESC;
```

Listing 4.3: Query to compute the number of dependency paths leading to vulnerable components with ”high” risk.

Empirical analysis on the SBOMs collected for this thesis revealed that the number of dependency paths leading to vulnerable components varies widely across projects, with typical values ranging from one to a handful of distinct paths. Although the path lengths from my experiments are are much smaller than the large propagation counts reported in the original VDGraph paper, the query still provides useful insight: even a small number of paths can show how a vulnerability is structurally introduced into a project.

Query 2: Vulnerability depth

A complementary question is how far vulnerabilities are in the dependency chain. This helps distinguish whether vulnerabilities arise from direct dependencies or from deep transitive ones.

```
MATCH p = (r:Root)-[:DEPENDENCY*]->(v:Vulnerability)
RETURN v.id AS vulnerability, min(length(p)) AS minDepth
ORDER BY minDepth ASC;
```

Listing 4.4: Query to compute the minimum dependency depth of each vulnerability

As in VDGraph, most vulnerabilities appear at depth greater than 2, confirming that analysing only direct dependencies is insufficient and that it is important to consider transitive dependencies too in these kind of studies.

Query 3: Most vulnerable components

Another useful analysis is to identify components that accumulate multiple vulnerabilities. This highlights libraries with recurring security issues or with historically fragile maintenance histories.

```
MATCH (c:Component)-[:HAS_VULNERABILITY]->(v:Vulnerability)
RETURN c.name AS component, count(v) AS numVulnerabilities
ORDER BY numVulnerabilities DESC
```

Listing 4.5: Query to retrieve components ordered by the number of associated vulnerabilities

This query revealed several libraries with multiple known vulnerabilities in the SBOMs submitted during the survey, making them natural priorities for review.

Query 4: Vulnerable leaf components

Leaf nodes are components with no other outgoing dependency edges. Identifying vulnerable leaves provides insight into whether vulnerabilities originate in terminal, low-level components or in higher-level, widely reused libraries.

```
MATCH (c:Component)
WHERE NOT (c)-[:DEPENDENCY]->(:Component)
MATCH (c)-[:HAS_VULNERABILITY]->(v:Vulnerability)
RETURN
  c.name AS component,
  collect(v.id) AS vulnerabilities
ORDER BY size(vulnerabilities) DESC;
```

Listing 4.6: Query to retrieve vulnerable leaf components in the dependency graph

Other queries

Besides the ones shown above, many other Cypher queries can also be run to study a project's vulnerabilities and dependencies. For example detecting components that introduce multiple vulnerable dependencies, computing the

longest dependency chains ending in vulnerable components, finding components with no vulnerabilities or filtering subgraphs by severity or ecosystem.

This analysis provides a support to the semantic queries made with SPARQL because it manages to provide a project-level and structural view that is not provided by SBOMs alone. These tools together contribute to a more comprehensive understanding of dependency risk and vulnerability propagation.

4.3.3 Function Graph Analysis

The GAT embeddings previously mentioned (Section 4.2.5) have been integrated with SBOM and `OSV` vulnerability data to compute a *risk score*, combining:

- GAT score, for structural importance.
- Betweenness centrality, for bridge behaviour.
- Degree, for local connectivity.
- Whether the function imports any vulnerable dependency.

This allows to identify functions that are both structurally central and directly affected by known external vulnerabilities.

As a sample repository I have taken CAiMIRA [8], a CERN-developed open source application for risk assessment used to model the concentration of viruses in enclosed spaces. The two tags under study are `v4.14.0` (base one) and `v4.15.0`. For example, functions such as `make_app` and `prepare_context` show high GAT scores and import modules with known vulnerabilities (`Jinja2`, `Tornado`, `urllib`). Even though these functions were not changed between the two versions studied, their position in the call graph suggests that they may require particular attention during dependency upgrades, since any behavioural change in the imported libraries could propagate through the code base.

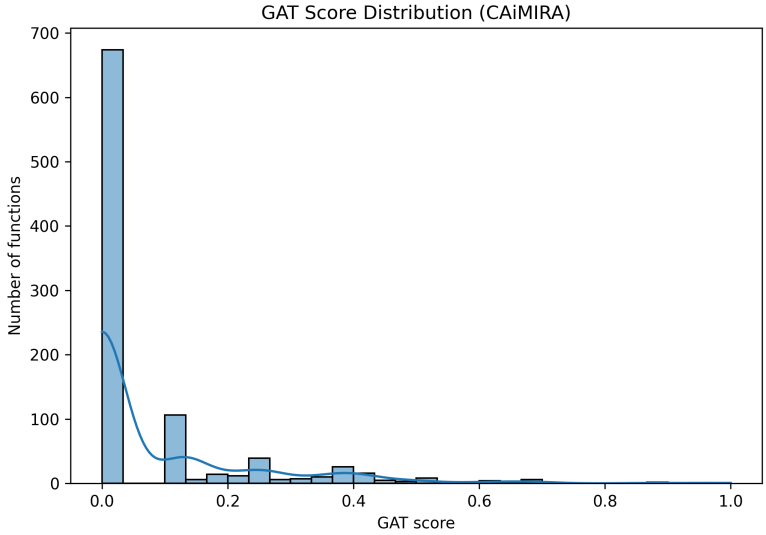

Figure 4.7: GAT score distribution for CAiMIRA.

Figure 4.7 illustrates the **GAT score distribution**, which is helpful to understand how structural importance is distributed. As expected, many functions have low scores meaning that they are not at the core of the application while only a few have high values, which represent key functions critical for the architecture.

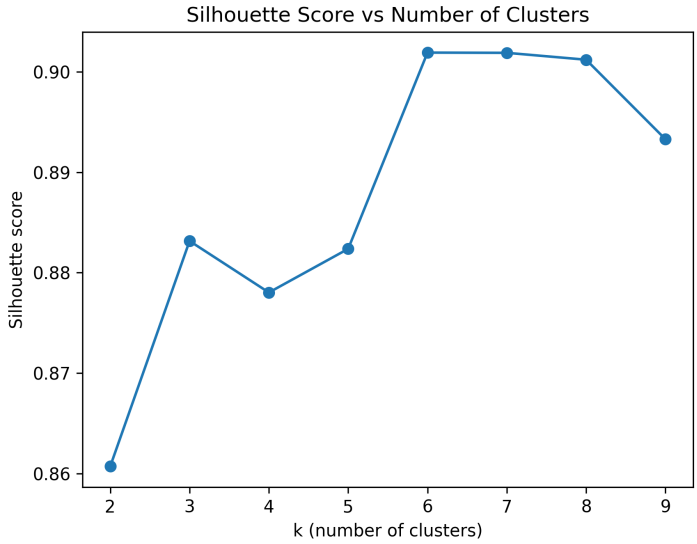

Figure 4.8: Silhouette score for CAiMIRA.

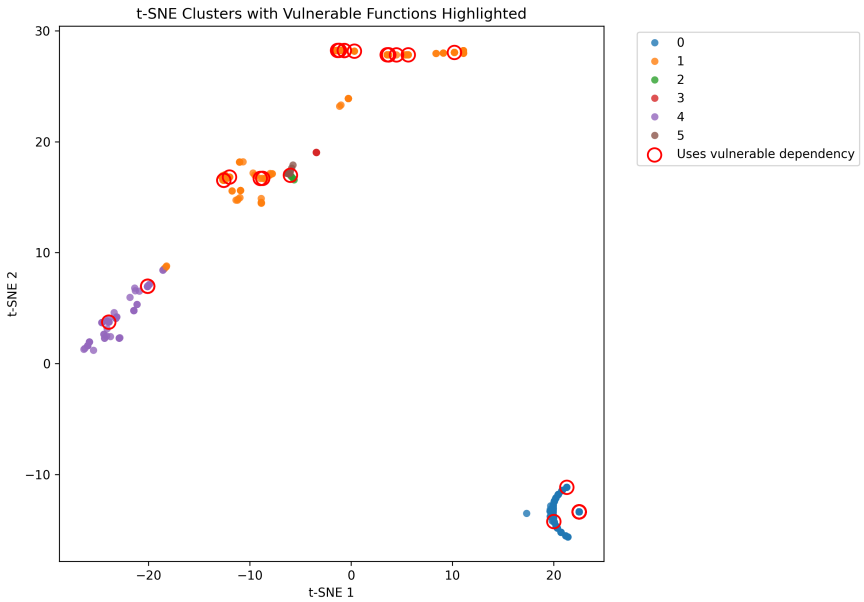

Figure 4.9: t-SNE projection of GAT embeddings.

To understand if the GAT embeddings capture meaningful information, also a **silhouette analysis** can be performed for varying values of k, as shown in Figure 4.8. All values are quite high so any of these number of clusters could be feasible but the most likely, with a score of 0.96, is $k = 6$ which suggests that CAiMIRA architecture naturally decomposes in six structural groups. In general, having high scores is a good indicator that the GAT is encoding neighbourhoods coherently and not randomly distributed [20].

This interpretation is further explored with a t-SNE projection of the embeddings, with the result being illustrated in Figure 4.9, where the six clusters are coloured by the result of the KMeans algorithm. The plot reveals clear separation among certain clusters and some dubious ones among other. The biggest one in the right corner is probably the ensemble of the orchestration scripts, while others could be testing code, documentation, domain-specific computations and so on. The functions using vulnerable dependencies are circled in red and is it can be seen do not form a cluster which is expected, since vulnerabilities could be in any part of the code and are not part a structural property of the graph.

Function	Risk	GAT	Deg	Betw.	Vuln IDs
`report/ virus_report.py: prepare_context`	0.912	0.134	4	2.75e-04	GHSA-cpwx-vrp4-4pq7 GHSA-q2x7-8rv6-6q7h GHSA-gmj6-6f8f-6699
`auth_service/ __init__.py: make_app`	0.639	0.111	4	3.65e-05	GHSA-7cx3-6m66-7c5m GHSA-8w49-h785-mj3c
`calculator/ __init__.py: make_app`	0.639	0.111	4	3.65e-05	GHSA-q2x7-8rv6-6q7h GHSA-7cx3-6m66-7c5m GHSA-8w49-h785-mj3c GHSA-cpwx-vrp4-4pq7 GHSA-gmj6-6f8f-6699
`calculator/ markdown_tools.py: _block_headings`	0.465	0.092	2	1.54e-06	GHSA-cpwx-vrp4-4pq7 GHSA-q2x7-8rv6-6q7h GHSA-gmj6-6f8f-6699
`calculator/ markdown_tools.py: extract_headings`	0.450	0.089	2	1.54e-06	GHSA-cpwx-vrp4-4pq7 GHSA-q2x7-8rv6-6q7h GHSA-gmj6-6f8f-6699
`calculator/ markdown_tools.py: extract_block`	0.450	0.089	2	1.54e-06	GHSA-cpwx-vrp4-4pq7 GHSA-q2x7-8rv6-6q7h GHSA-gmj6-6f8f-6699
`calculator/ markdown_tools.py: extract_rendered_ markdown_blocks`	0.445	0.088	2	1.54e-06	GHSA-cpwx-vrp4-4pq7 GHSA-q2x7-8rv6-6q7h GHSA-gmj6-6f8f-6699
`report/ virus_report.py: generate_permalink`	0.430	0.071	2	5.57e-05	GHSA-cpwx-vrp4-4pq7 GHSA-q2x7-8rv6-6q7h GHSA-gmj6-6f8f-6699
`tests/ test_webapp.py: end_time`	0.325	0.067	1	0.000	GHSA-7cx3-6m66-7c5m GHSA-8w49-h785-mj3c
`report/ virus_report.py: readable_minutes`	0.325	0.067	1	0.000	GHSA-cpwx-vrp4-4pq7 GHSA-q2x7-8rv6-6q7h GHSA-gmj6-6f8f-6699

Table 4.3: Highest risk score functions for CAiMIRA.

Finally, Table 4.3 lists the top ten functions ranked by risk score, computed as

$$0.6 * gat\_score + 0.3 * betwenness + 0.1 * degree$$

These functions are more likely to be grouped in high centrality regions and

import vulnerable dependencies. Notably, several such functions had not been modified between the two releases under analysis thereby underlining that functions with high structural significance, being at the core of the code, are more unlikely to be modified. Instead, the combination of GAT analysis and SBOM/OSV data can give good insights of the code base, highlighting what parts need more caution during dependency update.

Taken together, these findings suggest that analysis at the function level combined with GAT embeddings can identify architectural trends and possibly problematic hotspots not evident by analysing version-control data alone. While still at an experimental stage, the approach offers a promising complement to the SBOM and dependency-level analysis presented earlier in this thesis.

Chapter 5

Conclusions

5.1 Discussion

A key contribution of this thesis is the unification of three traditionally distinct viewpoints: semantic data modeling, vulnerability analysis at the dependency level and structural analysis of code, into one coherent workflow where every layer enhances the others.

The ontology-driven knowledge graph effectively integrates SBOMs, survey information, licensing data, and organisational details within a common semantic schema, enabling to answer complex queries that cannot be achieved by simply examining raw SBOMs. The second step, with VDGraph-oriented project analysis facilitates a structural insight into the propagation of vulnerabilities through dependencies, demonstrating how in actual CERN projects, the majority of vulnerabilities emerge at a depth greater than 2, reinforcing the importance of always taking into account transitive dependencies. Finally, function-level graph analysis indicate architectural patterns in actual codebases highlighting core functions, groups of related features, and regions most vulnerable to changes in external dependencies. When viewed collectively, these elements offer a more detailed insight into software supply chains, encompassing governance issues (such as licensing uniformity, dependency

usage across units) to security concerns (like which functions depend on at-risk libraries and the extent to which vulnerabilities are integrated into the structure).

Although its potential and results, the framework clearly presents some limitations. Most of all, the GAT-based analysis requires additional enhancement. The existing implementation shows that structural embeddings can reflect significant relationships in function-call graphs; however, at the moment the models depend on fairly basic features. The findings indicate that the concept is feasible, but further systematic experimentation across various languages, ecosystems, and more extensive datasets is essential for thorough validation. On this point, although the dataset utilized in the work is varied, it remains restricted in comparison to what CERN might offer. The survey gathered multiple SBOMs, and a few more were retrieved independently, yet the total number of real-world SBOMs examined is still quite limited in relation to CERN's software landscape. Ultimately, certain elements of the framework like connecting function-level details to the general ontology or combining reasoning across layers remain in a prototype phase.

A major opportunity for future development arises from the fact that specific departments at CERN, particularly the Accelerator Beams Department, currently have a pipeline for generating SBOMs and has a running inventory for them. These data are presently underutilized but, especially if further enhanced with CERN-specific metadata (Section 4.1) it could help enhancing the quality of studies on dependency and vulnerability propagation, it could significantly broaden the empirical basis of the framework, test the ontology on a level closer to the actual full organisation and much more. In other words, the data provided by this department has the potential for helping transform this project into a fully operational analytical tool, especially helpful in future iterations of the CERN Open-Source Software Dependency Survey (Appendix A).

An additional direction that could improve the analysis is the incorporation of CHAOSS project health metrics [14]. These metrics—like maintainer responsiveness, libyears (how much outdated the dependencies are), growth of issue backlog, community trends, or release frequency—provide crucial insights into risks related to ecosystem and maintainability that are not easy to obtain and of course cannot be derived directly from SBOMs by themselves. Integrating CHAOSS-based indicators with SBOM, OSV, and structural code information might, for instance:

- Emphasize dependencies that are both at risk and inadequately supported.
- Recognize elements experiencing reduced community backing.
- Supply preliminary alerts for concerns regarding software sustainability.
- Relate function-level risks to project-level governance elements

This would advance the analysis beyond just technical indicators and allow for a more comprehensive grasp of the supply chains strengths or weaknesses.

5.2 Conclusions

The study here presented shows how the integration of ontologies, dependency-graph modeling and code analysis can give real and practical insights into software supply chains. Every layer of this framework manages to present more detailed information; from the possibility of query SBOM data across the whole organisation regardless of the standard and origin of the document, to VDGraph that allows to uncover patterns in dependency utilization and vulnerability spread and to the function-level analysis that demonstrates how learned embeddings can identify architectural features otherwise hard to find.

Collectively, these elements show that a multi-layered strategy is both practical and advantageous for studying the software environment in a context like CERN. The framework already meets various practical requirements of OSPO officers, service managers, and Computer Security Team, covering areas like licensing consistency checks and evaluations of vulnerable dependency chains. At the same time, the research has underscored multiple aspects that need additional progress, in particular concerning the expansion of the dataset, enhancing the GAT-based analysis, and broadening the ontology to better interlink the various analytical layers.

Improvements going forward need to focus on validating the function-level module on wider and more diverse code bases. The integration of CHAOSS metrics would broaden the analytical framework, allowing for evaluations that merge technical data with organisational and community indicators, thereby enhancing the quality of the insights generated.

In conclusion, this thesis offers a useful, versatile and extensible basis for analysing supply chains in large and heterogenous environments. Through ongoing enhancement and more comprehensive data integration, the framework can develop into a strong operational tool that aids informed governance and contributes to the long-term sustainability of open-source software at CERN and beyond.

Appendices

Appendix A

CERN Open-source Software Dependency Survey

This appendix describes the CERN Open-source Software Dependency Survey –hereafter simply *the Survey* – which serves as one of the data sources utilized in this thesis. The questionnaire was sent out to the main organisational units engaged in software development at CERN, requesting **service managers** to declare the applications they oversee along with their key dependency sets. Participants had the option to give written responses or submit SBOMs enriched with CERN-specific metadata through a tool created for this purpose.

First of all, in *Section 1*, respondents managers were asked to choose whether to upload SBOMs or manually fill the Survey. If they chose the *SBOM* option, they were redirected to a page with the following fields:

Section 2 - SBOM

SBOM (required)

If you want to try to generate an SBOM you can follow the guidelines present in the following link: *Recommendations for SBOM generation [9]*.

Review it and **check if the mandatory fields are present** (contact email, organisational unit, Service-now element, application name and URL), otherwise add them manually as specified in the previous link.

Maximum number of file uploads is 10 per form; in case you have more SBOMs to submit, simply fill the survey again.

Upload SBOM file(s):

Comments

If you encountered any issues during SBOM creation or have any other comments, please indicate them here.

If, otherwise, they preferred to fill the Survey manually they were presented with the following:

Section 3 - General information

Your email or a reference e-group:

Organisational unit (Department–Group–Section):

Service element in Service-now (if applicable).

Section 4 - Application

Application name.

Example: CodiMD, Twiki...

Please provide a link to the source code for the main open-source package used in your service.

Please indicate any other relevant open-source packages used in your service (separated by commas if more than one).

Would you like to add another application?

○ Yes ○ No (Submit)

Section 4 could be filled up to ten times by service managers within the same response, in order to simplify the work for service managers declaring multiple applications.

Bibliography

[1] G. Antoniou and F. v. Harmelen. Web ontology language: owl. In *Handbook on ontologies*, pages 91–110. Springer, 2009.

[2] V. Arraj. Itil®: the basics. *Buckinghampshire, UK*, 2010.

[3] D. A. Bader and K. Madduri. Parallel algorithms for evaluating centrality indices in real-world networks. In *2006 International Conference on Parallel Processing (ICPP'06)*, pages 539–550. IEEE, 2006.

[4] N. Baken. Linked data for smart homes: comparing rdf and labeled property graphs. In *LDAC2020—8th Linked Data in Architecture and Construction Workshop*, pages 23–36, 2020.

[5] S. Bechhofer and A. Miles. Skos simple knowledge organization system reference. *W3C recommendation, W3C*, 2009.

[6] T. Berners-Lee et al. Semantic web road map, 1998.

[7] T. Berners-Lee, J. Hendler, and O. Lassila. A new form of web content that is meaningful to computers will unleash a revolution of new possibilities. *Scientific american*, 284:05–01, 2002.

[8] CERN. Caimira. `https://github.com/CERN/CAiMIRA`, 2025.

[9] CERN OSPO recommendations for sbom generation. `https://ospo.docs.cern.ch/howtos/generating-sbom/`.

[10] D. Chen, J. Chen, X. Zhang, Q. Jia, X. Liu, Y. Sun, L. Lü, and W. Yu. Critical nodes identification in complex networks: a survey. *Complex Engineering Systems*, 5(3), July 2025. ISSN: 2770-6249. DOI: 10.20517/ces.2025.34. URL: http://dx.doi.org/10.20517/ces.2025.34.

[11] D. M. Clavo, B. Brugger, N. Cremel, C. Delamare, I. F. Gonzalez, E. Lienard, R. Martens, M. Moller, W. Salter, Z. Toteva, et al. Challenges, solutions and lessons learnt in 7 years of service management at cern. In *EPJ Web of Conferences*, volume 214, page 08003. EDP Sciences, 2019.

[12] CVE — common vulnerabilities and exposures. https://cve.org. Accessed: 2025-11-25.

[13] C. Ebert, J. Cain, G. Antoniol, S. Counsell, and P. Laplante. Cyclomatic complexity. *IEEE software*, 33(6):27–29, 2016.

[14] S. Goggins, K. Lumbard, and M. Germonprez. Open source community health: analytical metrics and their corresponding narratives. In *2021 IEEE/ACM 4th International Workshop on Software Health in Projects, Ecosystems and Communities (SoHeal)*, pages 25–33. IEEE, 2021.

[15] K. Goseva-Popstojanova and A. Perhinschi. On the capability of static code analysis to detect security vulnerabilities. *Information and Software Technology*, 68:18–33, 2015.

[16] T. Gruber. What is an ontology, 1993.

[17] M. Gruninger. Methodology for the design and evaluation of ontologies. In *Proc. IJCAI'95, Workshop on Basic Ontological Issues in Knowledge Sharing*, 1995.

[18] ISO/IEC 19770-2:2015 information technology — it asset management — part 2: software identification tag. International Standard, 2015. ISO/IEC 19770-2:2015.

[19] B. Khemani, S. Patil, K. Kotecha, and S. Tanwar. A review of graph neural networks: concepts, architectures, techniques, challenges, datasets, applications, and future directions. *Journal of Big Data*, 11(1):18, 2024.

[20] D. Knežević, J. Babić, M. Savić, and M. Radovanović. Evaluation of lid-aware graph embedding methods for node clustering. In *International Conference on Similarity Search and Applications*, pages 222–233. Springer, 2022.

[21] M. Krötzsch. Owl 2 profiles: an introduction to lightweight ontology languages. In *Reasoning Web International Summer School*, pages 112–183. Springer, 2012.

[22] A. M. Mir, M. Keshani, and S. Proksch. On the effect of transitivity and granularity on vulnerability propagation in the maven ecosystem, 2023. arXiv: `2301.07972 [cs.SE]`. URL: `https://arxiv.org/abs/2301.07972`.

[23] National Vulnerability Database (NVD). `https://nvd.nist.gov`. Accessed: 2025-11-25.

[24] G. Navarro. A guided tour to approximate string matching. *ACM computing surveys (CSUR)*, 33(1):31–88, 2001.

[25] Neo4j graph database. `https://neo4j.com`. Accessed: 2025-11-25.

[26] A.-C. N. Ngomo and S. Auer. Limes-a time-efficient approach for large-scale link discovery on the web of data. *integration*, 15(3), 2011.

[27] A.-C. Ngonga Ngomo, M. A. Sherif, K. Georgala, M. Hassan, K. Dreßler, K. Lyko, D. Obraczka, and T. Soru. LIMES - A Framework for Link Discovery on the Semantic Web. *KI-Künstliche Intelligenz, German Journal of Artificial Intelligence - Organ des Fachbereichs "Künstliche Intelligenz" der Gesellschaft für Informatik e.V.*, 2021. URL: `https://papers.dice-research.org/2021/KI_LIMES/public.pdf`.

[28] E. O'Donoghue, Y. Hastings, E. Ortiz, and A. R. M. Muneza. Software bill of materials in software supply chain security a systematic literature review, 2025. arXiv: 2506.03507 [cs.SE]. URL: https://arxiv.org/abs/2506.03507.

[29] OSV — open source vulnerabilities. https://osv.dev. Accessed: 2025-11-25.

[30] Osv-scanner. https://github.com/google/osv-scanner. Accessed: 2025-11-25.

[31] OWASP Foundation. https://owasp.org. Accessed: 2025-11-25.

[32] OWASP Foundation. Cyclonedx specification, version 1.4. https://cyclonedx.org/specification/1.4/, 2023. Accessed: 2025-11-06.

[33] S. Sparks, S. Embleton, R. Cunningham, and C. Zou. Automated vulnerability analysis: leveraging control flow for evolutionary input crafting. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pages 477–486. IEEE, 2007.

[34] SPDX Workgroup, Linux Foundation. Spdx specification, version 2.3. https://spdx.github.io/spdx-spec/v2.3/, 2022. Accessed: 2025-11-06.

[35] K. Stewart, P. Odence, and E. Rockett. Software package data exchange (spdx) specification. *IFOSS L. Rev.*, 2:191, 2010.

[36] H. Stuckenschmidt, C. Parent, and S. Spaccapietra. *Modular ontologies: concepts, theories and techniques for knowledge modularization*, volume 5445. Springer, 2009.

[37] R. Studer, V. R. Benjamins, and D. Fensel. Knowledge engineering: principles and methods. *Data & knowledge engineering*, 25(1-2):161–197, 1998.

[38] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

[39] F. Vera, P. Pauliuchenka, E. Oh, B. C. Kao, L. DiValentin, and D. A. Bader. Profile of vulnerability remediations in dependencies using graph analysis. *arXiv preprint arXiv:2403.04989*, 2024.

[40] H. Xia, J. Gluck, S. Simsek, D. S. Medina, and D. Starobinski. Vdgraph: a graph-theoretic approach to unlock insights from sbom and sca data. *arXiv preprint arXiv:2507.20502*, 2025.

[41] E. Yangulova, M. Hirsch, S. Holland, and D. Iglezakis. Software License Ontology, version V1, 2025. DOI: `10.18419/DARUS-4738`. URL: `https://doi.org/10.18419/DARUS-4738`.

[42] K. Zhu, Y. Lu, H. Huang, L. Yu, and J. Zhao. Constructing more complete control flow graphs utilizing directed gray-box fuzzing. *Applied Sciences*, 11(3):1351, 2021.

Acknowledgements

I would like to thank professor Chesani for really helping me out while writing this thesis and motivating me.

Thanks to Giacomo for believing in me and entrusting me with this project, for the continuous support, and for being the best guide I could have asked for.

Thanks to my parents and Baltito for always being there, their patience and for always supporting me.

Thanks to my life-long friends in Cesena, who have been an anchor during these years and made them easier.

Thanks to the friends from every corner of the world I made during my time at CERN both as Technical Student and Summer Student, with whom I spent the best time.

Thanks to the friends I made during my Erasmus in Θεσσαλονίκη, for all the freddo espressos and the spanakotiropites.

I am grateful to whoever has been by my side during the journey I went through during this master's degree.